AD-A196 115

UNCLASSIFIED

DTIC FILE COPY

(1)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/CI/NR 88-68	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A FAULT TOLERANT SELF-ROUTING COMPUTER NETWORK TOPOLOGY		5. TYPE OF REPORT & PERIOD COVERED PHD ~~MS~~ THESIS
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) TONY L. MITCHELL		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFIT STUDENT AT: NORTH CAROLINA STATE UNIVERSITY		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE 1988
		13. NUMBER OF PAGES 268
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) AFIT/NR Wright-Patterson AFB OH 45433-6583		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

DISTRIBUTED UNLIMITED: APPROVED FOR PUBLIC RELEASE

DTIC ELECTE AUG 0 3 1988 S D

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

SAME AS REPORT

18. SUPPLEMENTARY NOTES

Approved for Public Release: IAW AFR 190-1
LYNN E. WOLAVER [illegible] 19 July 88
Dean for Research and Professional Development
Air Force Institute of Technology
Wright-Patterson AFB OH 45433-6583

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ATTACHED

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

A Fault Tolerant Self-Routing Computer Network Topology

by

Tony L. Mitchell

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Department of Electrical and Computer Engineering

Raleigh

1987

Approved by

Co-Chairman of Advisory Committee

Co-Chairman of Advisory Committee

BIOGRAPHY

Tony L. Mitchell is a Lieutenant Colonel in the United States Air Force. He is currently an Assistant Professor of Mathematics in the Department of Mathematical Sciences at the U. S. Air Force Academy in Colorado Springs, Colorado.

PII Redacted

Colonel Mitchell was born in [redacted] [redacted] [redacted] [redacted] [redacted] He received his military commission in 1970, as a Distinguished Graduate of the Air Force Reserve Officers Training Program at North Carolina Agricultural and Technical State University in Greensboro, N. C. Military assignments include Base Communications Officer at Loring Air Force Base, Maine (1971 - 1974), Computer Systems Analyst at Tinker Air Force Base, Oklahoma (1976 - 1978), Communications Detachment Commander at Izmir, Turkey (1978 - 1980), and Instructor of Mathematical Sciences at the United States Air Force Academy (1980 -1982).

Colonel Mitchell's professional interests include computer communications, digital architecture, software systems, and applied mathematics. He has a B. S. in Mathematics from North Carolina Agricultural and Technical State University (1970), and an M. S. in Information and Computer Science from the Georgia Institute of Technology in Atlanta (1975). Tony and his wife, the former Carolyn Mosley of Greenwood, S. C., have three sons, Anthony, Kevin, and Gary.

ACKNOWLEDGEMENTS

For his guidance, patience, and encouragement, I thank my advisor, Dr. Arne A. Nilsson. I also express my appreciation to Dr. Wushow Chou for his early and continued understanding and support. I am especially grateful to my lovely wife, Carolyn, for her constant encouragement and strong belief that God would see us through the most challenging years of our marriage. Most importantly, I thank God for His sustaining grace during the past five years.

TABLE OF CONTENTS

LIST OF FIGURES

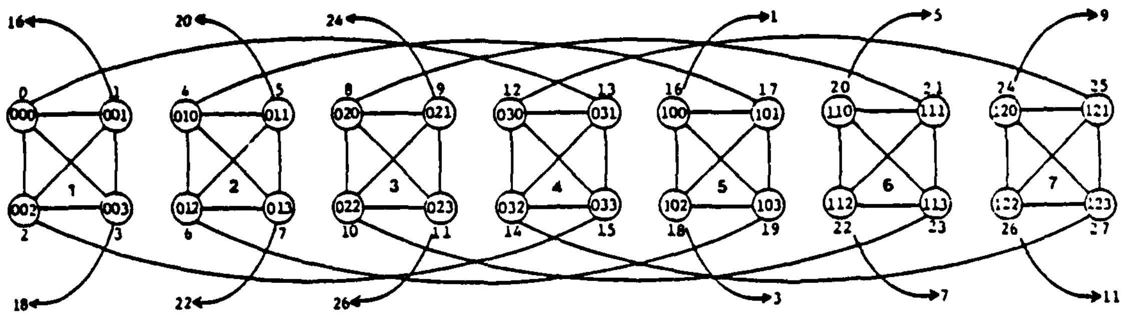
LIST OF TABLES

1. INTRODUCTION

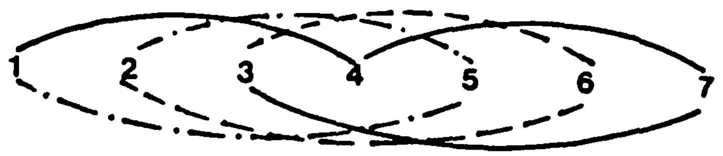
This document reports on the development and analysis of a new, easily expandable, highly fault tolerant self-routing computer network topology. The topology applies equally to any general purpose computer networking environment. This new connectivity scheme is named the "spiral" topology for reasons that will become evident shortly. A spiral network evolves by adding modules to a minimum starting topology. A module consists of four fully connected computer nodes. The modules are added to the existing topology one at a time, until the desired network size is attained. Figure 1-1(a) shows an example of a seven module, 28 node network. The module threading pattern in figure 1-1(b) depicts more clearly the order in which modules are connected.

The spiral topology features a simple internal self-routing algorithm that adapts quickly, and automatically, to failed links or nodes. The simple routing feature, our selection of four computer nodes to form a module, and the ease at which the network can be expanded, are all direct consequences of our choice of a base four (4) node numbering system for internal control of the network.

Analysis confirms that throughout the spiral network, routing can be done "on-the-fly" based on spiral and direction flags initialized at the source node. No global



a). Seven Module Spiral Network.



b). Threading Pattern.

Figure 1-1. Seven Module Spiral Network Topology.

network connectivity or routing tables are necessary at nodes. There is no global network overseer (master) node since all nodes operate at the same common precedence level. A particular node has routing and status knowledge of directly connected nodes only.

The spiral topology, with its fast on-the-fly routing capability, is highly amenable to fiber optic communications in both local and wide area computer networks. The fast routing attribute precludes storing of messages while a routing decision is being made. Further, since the simulation model assumes, and is run under Poisson arrival rate of exponentially distributed messages, Integrated Services Digital Network (ISDN) type traffic can be passed over the network with ease. This is possible because ISDN traffic is expected to be an independent mix of voice, data, and video that will be transmitted using circuit and packet switching technology. A Poisson arrival rate of traffic means that the time between message arrivals is exponentially distributed, and the independent traffic mix of voice, data, and video is best modeled by selecting the size of transmitted messages from an exponential distribution. Assuming a fixed transmission rate, the service time of this traffic mix will be exponentially distributed.

The motivation for a new computer network topology is introduced in chapter 2, where two current applications areas are discussed. In chapter 3, we present a brief description of the four traditional network topologies: the bus, ring, star, and mesh. Chapter 4 consolidates the time delay formulas for the most popular local area network topologies.

Chapter 5 marks the beginning of the new and significant contribution to the field of Computer Communications, since that chapter contains a general description of the spiral network topology. Chapter 6 addresses how the spiral routing algorithm operates with and without failed computer nodes. A detailed analysis of the error free spiral network is presented in chapter 7, and chapter 8 contains the results of analysis when nodes have failed. Chapter 9 summarizes conclusions resulting from a thorough study of the spiral network topology. And finally, Chapter 10 contains a discussion of suggested areas for additional study.

2. MOTIVATION FOR A NEW TOPOLOGY

2.1. Specific Operational Requirement

The National Aeronautics and Space Administration (NASA) is currently pursuing its objective of having a Space Station operational in the 1990's. NASA planners foresee the station evolving from an initial physical "module" [1] that supports critical life support and spacecraft navigational functions, to a fully manned self-sustaining station where several dozen space pioneers live and work for extended periods of time. The station will evolve one module at a time, until the complete desired configuration is attained [1].

NASA expects each module to contain 4 - 5 computer nodes that must interact and be highly tolerant to faults within that module, and throughout the entire network. A fault is defined as a failed network node or link. Limited physical space within each station module, and exorbitant cost, preclude hardware duplication as the answer to the fault tolerance requirement. Further, NASA envisions the station evolving to a total of 10 - 20 modules, each containing 4 - 5 local area computer network nodes [1]. NASA requires that all nodes have access to all others, no matter where they are physically positioned in the Space Station.

Although the station is not expected to become operational until the 1990's, Space Station planners and developers have already defined fairly clearly the type of environment, and communications needs expected in that environment. In particular, a Space Station Information System [48] has been described, and functions of the various components have been allocated to specific data system building blocks.

The following list documents the anticipated functions of the Space Station Information System [48]:

- Manage Customer/Operator Delivered Data
- Manage Customer/Operator Supplied Data
- Schedule and Execute Operations
- Operate Core Systems
- Manage Facilities and Resources
- Develop, Simulate, Integrate, and Train
- Support Space Station Programs

It can be seen from this list that the "users" of the Space Station Information System will include Space Station core systems, crew members, as well as customer equipment.

Although subject to change as technological advances and constraints are realized, the following general functions of the Command and Data Management Support Subsystem for the Space Station are expected to mold the type of communications network ultimately deployed aboard the station [1]:

* Communications and Tracking
  - Voice intercom
  - Detached vehicle
  - Ground crew and vehicles
* Data Handling
  - Acquisition/Retrieval of data
  - Data distribution (automatic and upon demand)
  - Data processing/number crunching
  - Storage of data
  - Realtime support for display and crew input
* Closed-Circuit TV
  - Cameras and monitors
* Timing
  - Generation
  - Distribution
  - Timing displays

These general functions must be implemented in a fashion that is both efficient, and user friendly. Further, a key aspect of the resulting communications network is that it be highly fault tolerant. In an environment as isolated from the earth as the Space Station will be from a logistics viewpoint, highly reliable fault tolerant communications among those aboard the station, as well as between the station and ground control, is essential.

Therefore, NASA has identified the following preliminary requirements of the communications network that will form the heart of the Space Station Data System [1]:

* Highly fault-tolerant
* Having between ten and one hundred nodes
* All nodes are created equal, but not all messages
* The network will operate in the Space Station composed of from ten to twenty space lab modules, with four to five nodes per module
* Total Space Station configuration has the maximum dimension of 150 feet
* The space lab modules are butted up against each other to form complex shapes
* The network management functions will be distributed among the nodes

Based on these requirements, it becomes quite clear that a local area network with standard interfaces (and identical equipment where possible), and high bandwidth capability is necessary to provide both responsiveness and the high degree of fault tolerance mandated. Of course the degree of fault tolerance is more a function of the connectivity of the resulting network, than it is the speed as perceived by the user.

To meet these requirements, NASA needs a network topology that is 1) highly tolerant to failure(s); 2) easily expandable without having to reconfigure the existing topology; 3) capable of operating via decentralization of control; and 4) capable of handling an integration of voice, video, and data traffic. It will be shown that the spiral topology meets these needs.

2.2. Integrated Services Digital Network

2.2.1. Introduction

This section presents a brief overview of the Integrated Services Digital Network (ISDN). Specific transmissions protocols that can be used for integrating voice and data in a local area network are addressed in [2]. Further, a detailed analytical discussion of several mathematical models suitable for integrating voice and data is presented by M. Schwartz in [3].

Information networks around the globe are approaching a revelation in new service and revenue opportunities. The revelation is ISDN. The opportunities derive from the transport power and flexibility of ISDN's standard access interfaces and integrated channel structures. End users, network providers, and network-systems suppliers alike will benefit from a host of integrated voice and data services which ISDN makes possible for the first time [4].

Evolution of the public-switched telephone network in the past has been governed primarily by the need to provide voice services. The network that evolved was analog, and predominately electromechanical. Before 1962 (when pulse code modulation was introduced), the transmission facilities for the telephone network were all analog [5, 6]. This analog network was not well suited to serve the emerging needs of data, facsimile, and video. In the past decade, however, the many advantages of digital systems (small cost and size, large transmission and switching capabilities, high signal quality, flexibility, ease of maintenance) have promoted the conversion of transmission and switching systems from analog to digital in many telephone administrations around the world.

The criteria for justifying conversion to digital technology have been lower life-cycle operating costs rather than increased revenue due to new nonvoice services. So although the switched network is slowly being transformed to a digital one, the architecture of the facilities is still biased towards providing voice service [6].

The adoption of digital techniques in the public switched network makes it possible and cost effective to integrate voice, data and video services into a single Integrated

Services Digital Network. The ISDN is envisioned as an international digital communications network, supporting a wide spectrum of user needs [8]. The central theme in ISDN is that different services, all of them digital in nature, can use this same connection or channel, resulting in better channel utilization.

The fundamental principles embodied in an ISDN are [7] digitization with high bandwidth, world-wide standards, and integrated fuctionality and services. The three elements generally recognized as necessary to support these principles are [3]: 1) All digital channels are used end-to-end; 2) the network handles a multiplicity of services with possibly differing bandwidths using interleaved bit streams; and 3) there are standard interfaces for user access.

The major bottleneck impacting the evolution to an ISDN appears to be the conversion of existing subscriber local loops and equipment, to digital operation [9, 10].

2.2.2. Definition of an IDSN

The goal of the ISDN is to provide a versatile, multiservice network with standard customer interfaces and international capabilities. This goal results in the definition and standardization of digital interfaces, both user-to-network, and network-to-network. ISDN is a generic term referring to the integration of communication services transported over digital facilities. As a public network concept, ISDN deals with the evolution of the digital network as a carrier of both voice and data applications [8, 11]. The ISDN aim is to provide cost-effective, end-to-end digital connectivity to support a wide range of voice and nonvoice services [12]. The International Telegraph and Telephone Consultation Committee's (CCITT) conceptual principle is that "The ISDN will be based on and evolve from the telephony ISDN by progressively incorporating additional functions and network features, including those of any other dedicated networks such as data packet-switching, so as to provide for existing and new services" [6]. The CCITT defines ISDN as

> A network evolved from the telephony Integrated Digital Network (IDN) that provides end-to-end digital connectivity to support a wide range of services, including voice and nonvoice services, to which users have access by a limited set of standard multipurpose user-network interfaces.

ISDN is motivated by the economics and flexibility associated with multiservice applications. The evolution to multiservice applications has led to the ISDN concept of a family of standard customer interfaces which provide access to the network, all of which operate in synchronous, full duplex mode [6].

The standard interface concept is being vigorously addressed by the CCITT. Study groups of the CCITT are working on the development of interfaces that will be compatible with existing 64-kbps digital voice channels, and that will incorporate signaling channels as well. An example of such an interface, the first one developed as a CCITT recommendation [3], is the 2B + D narrowband interface. This consists of two 64-kbps B channels for information transfer, and a 16-kbps D channel for signaling and other uses (figure 2-1). The three channels, totaling 144-kbps of transmission capability, are interleaved. A B channel could be used for digital voice or circuit switched data; the D channel could be used for carrying packet switched data as well as control packets [3]. In particular, the D channel may be used for signaling for the B channel, telemetry, and low-speed data transport [11]. Wider band interfaces, based on an nB + D structure, with the D channel considered a 64-bkps channel, are also being developed as CCITT recommendations. With n = 23, this possible standard makes the structure compatible with the 1.544-Mbps T1 standard; n = 30 makes it compatible with the worldwide 2.048-Mbps digital transmission standard. Other interface recommendations are designed to handle even higher bandwidth services such as video and high speed facsimile [3].

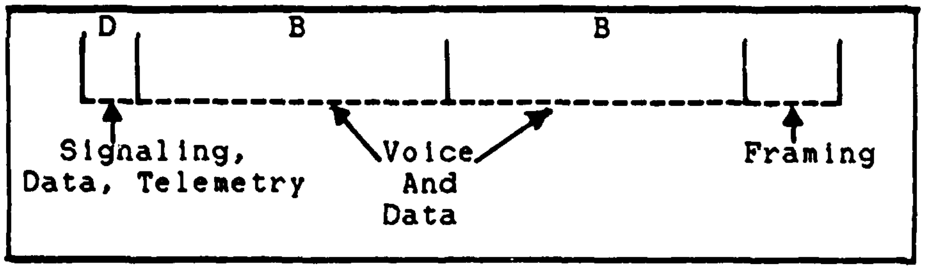


Figure 2-1. Concept of Potential ISDN Frame.

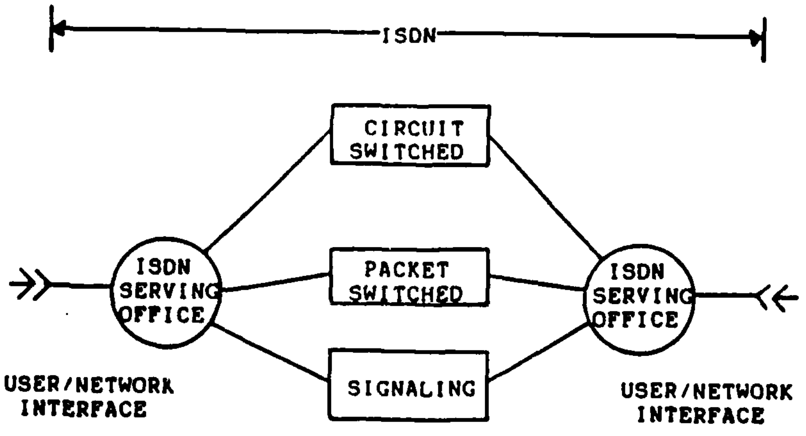


Figure 2-2. Example of ISDN Capabilities.

In the Bell Operating Companies (BOC) networks, initial ISDN integration will be realized on the access portion of the network. Figure 2-1 displays the concept of an ISDN interface frame. And figure 2-2 represents the conceptual view of ISDN capabilities [11].

2.2.3. Architecture

Irvin Dorros [13, 14] suggests the general local architecture shown in figure 2-3. The key ISDN concepts are that the customer will be supplied with a telecommunications transport capacity measured in maximum bit rate at a standardized interface. This bit stream capacity will be provided by the network to a customer's premise in what Dorros calls a "digital pipe". The customer will aggregate the variable bit rate capacity needs of his terminals at a control device that interfaces with this network pipe. Ultimately, packet and circuit switching access will be integrated and provided on the same pipe.

As an architectural representation for the network itself, Mario Gerla [5] forsees ISDNs evolving in three phases: In phase I, the ISDN will consist of two separate networks, one for circuit-switched (CS) traffic, and the other for packet-switched (PS) traffic, and a single access interface for both networks. In phase-II, CS and PS traffic will share transmission media but will use different switching

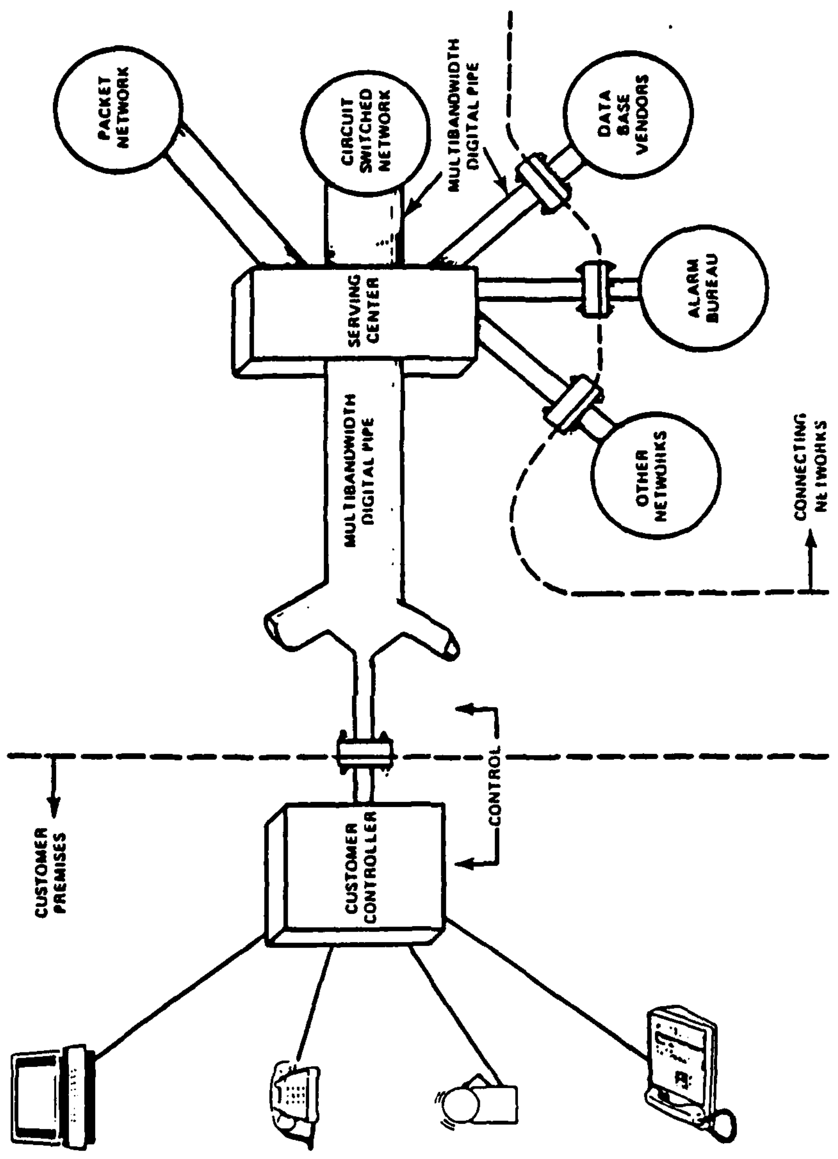


Figure 2-3. ISDN Local Access.

facilities. Finally, in phase III, transmission and switching facilities will be shared by CS and PS traffic in a fully hybrid integrated network.

Perhaps the most elaborate description of the evolving ISDN architecture is contained in "Bell's Concept of the ISDN" [15]. This reference provides three separate Bell system network capabilities, each of which can support one or several ISDN applications. All of these capabilities provide switched, end-to-end digital, full-duplex data connections over 2-wire loops. The first capability deals with circuit switching of 56-kbps. The second deals with packet switching in a local area at speeds of up to 8-kbps and possibly higher. This second capability involves the use of statistical multiplexers to concentrate many calls onto a single 56-kbps line. The final capability deals with long haul packet switching at speeds of 56-kbps.

An outline of current proposals for an early application of optical fiber in the local network, and discussion of how these proposals act as stepping stones toward the broadband ISDN is found in [16].

2.2.4. Service Requirements and Network Capabilities

ISDN planning must take into consideration the wide range of customer service requirements that an ISDN might be required to support. These customer needs include [15] interactive data, image processing, bulk data, audio and video. Further, standard ISDN user-network interfaces must be responsive to these customer needs [18]. The following sections give some general characterizations of the transmission bit rates of these customer needs. Other important considerations include the burstiness of traffic, the error rate required and session length. All of these vary over a wide range depending upon the application. Figure 2-4 graphically summarizes some of these characteristics.

Many telemetry type applications such as meter reading, energy management, and security, have very low average bit rates, less than 300-bps. Interactive data applications generally use terminals to access data bases, word processors, computers and other terminals. These applications include inquiry/response and transactions which tend to have bit rates less than 4.8-kbps [15].

Image applications are characterized by the transmission of fixed images and include facsimile, graphics and slow-scan or freeze-frame TV. They can be handled with a bit rate of

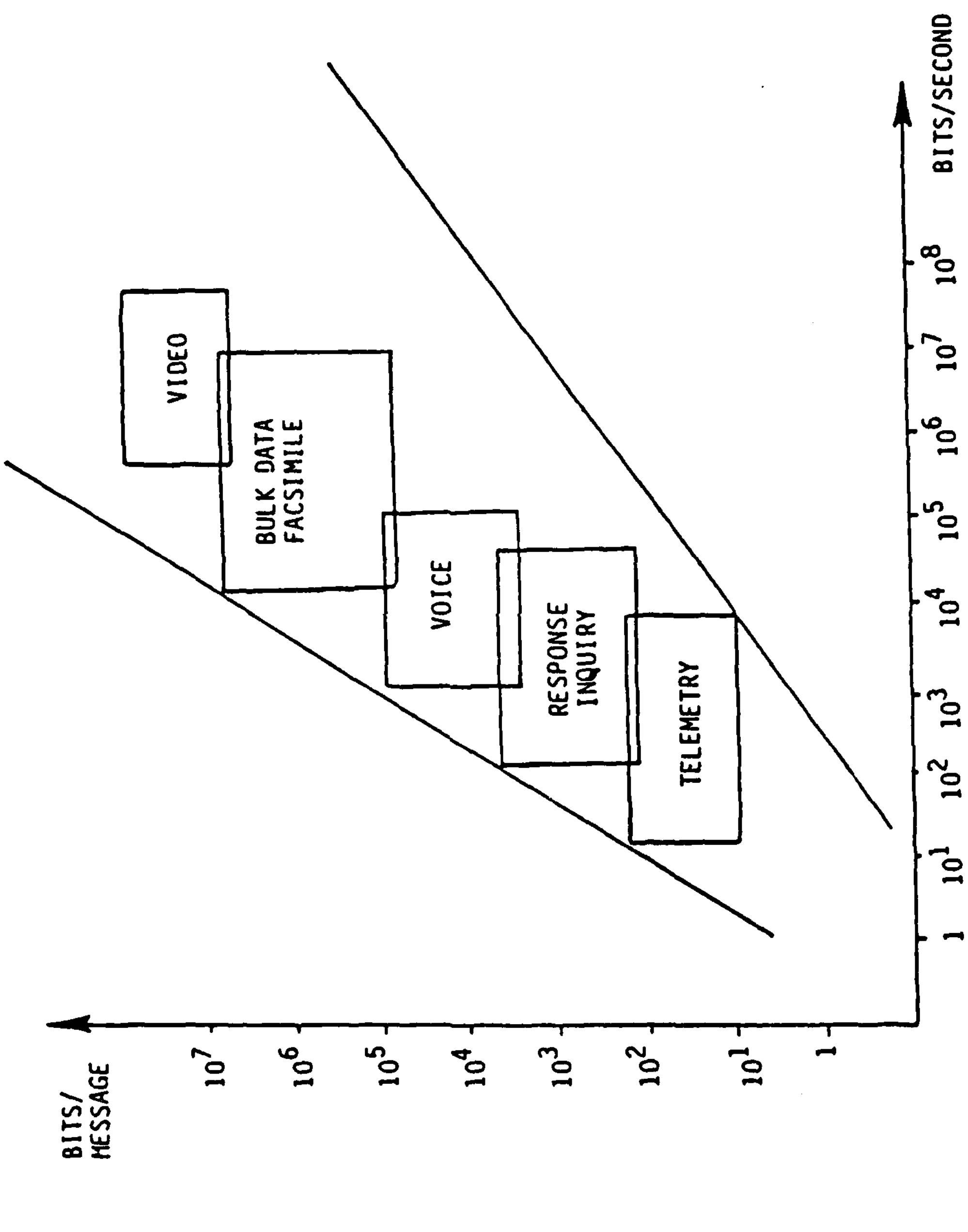


Figure 2-4. Characteristics: User Needs.

64-kbps or less [15].

Audio includes signals such as voice and music. In the U.S., voice has generally been encoded at 64-kbps for transmission on the network. Methods do exist, however for transmission of voice at speeds less than 64-kbps with acceptable quality for many applications [15].

Bulk data transfer encompasses applications such as the transmission of large data files between computers. One application is the nightly transfer of billing data from large remote locations to a central host facility. Such applications can use speeds of up to 1.544-Mbps [15].

Full motion video can be provided in a number of formats. For example, broadcast quality video requires about 4.5-MHz as an analog signal or approximately 100-Mbps as a digital signal. Of course signal compression techniques can reduce this considerably [10, 15].

The various customer needs just described can be generated from a wide variety of customer types [17]. These include the banking and airline industries, universities, electronic publishing, electronic suppport of catalog shopping, automated offices, and even private networks that serve the specific needs of single large customers.

Studies of the ISDN objectives have identified four general categories of network-level functional entities that may be needed [12]:

* Transaction routing and control
* Network management and operation
* "Add-on" characteristics (i.e. service options)
* Information processing

2.2.5. Evolution to an ISDN

Several authors express their own interpretation of how ISDN has and will continue to evolve [5, 10, 11, 12, 14, 15, 17, 19, 21]. Contained in this section is an attempt to tie these approaches together in a way to demonstrate that although unique interpretations exist, they all tend to point in the same general direction. This section discusses a global view of ISDN as key world-wide efforts are cited. A recent issue of the Journal on Selected Areas in Communications [20] contains numerous articles by various authors on how different ISDN system components will likely be implemented. And finally, Appendix A contains further details on the evolution to an ISDN, addressing the Bell Operating Companies (BOCs), and specific subsets of the global ISDN.

ISDN evolution will not be unique [12, 14], since transformation will vary depending on such factors as the existing network, operating organization, regulation, competition, geographical and economic environment, and technology and interpretation of standards.

ISDN's will be based on the concepts developed for telephone Integrated Digital Networks (IDN's), and may evolve by progressively incorporating additional functions and network features, including those of other dedicated networks such as circuit-switching and packet-switching for data, so as to provide for existing and new services [6, 12].

Information users are looking to ISDN as a future telecommunications service shopping center [12]. Shopping centers yield efficient, easy access, and economies of scale, by bringing many services under one roof. The ISDN is being designed to give users a uniform view of a wide variety of applications.

Although cost-effective functionality, and not technology, is the user's primary concern [12], users realize benefits of an ISDN. They need timely, stable standards in order to make cost-effective plans to tailor their network's evolution to their business demands. The CCITT layered interface approach to ISDN standardization responds to the user's needs, since it permits terminal and network technologies to evolve independently. ISDN flexibility is achieved through setting digits in signaling protocol fields and not setting voltages on a large number of interface wires [12].

We now look at where we are in achieving the ISDN vision [14]. General concensus within the industry is that for integrated services networks, voice will be dominant for a long time to come and will be the factor which sets the pace of a truly integrated evolution. In the U.S., we have surpassed the 1000 time division central office mark. In addition, there are thousands of digital Private Branch Exchanges (PABX's) of various sizes which connect into our national communications network.

In transmission, there are nearly 100 million circuit miles of T-carrier (T1, T2, etc), and over 5 million circuit miles of digital radio in the U.S. There are approximately a quarter of a million loops employing digital subscriber carriers. Lightwave is also a fast emerging digital tramsmission technology [14].

An important capability in an evolving ISDN is common channel signaling. Over 25% of U.S. intertoll trunks are now utilizing the world's largest packet switched network - the Common Channel Interoffice Signaling (CCIS) Network [14]. Thus in the U.S., we have been and will continue to introduce the transmission, signaling, and switching parts of the ISDN.

The TransCanada Telephone System is developing a fundamental plan for an ISDN through 1990. By the late 1980's, about 2 million Canadian lines are expected to be served from digital central offices. Canada already has an extensive packet switched network called DATAPAC with interconnections to other countries [14].

There can be little doubt about the commitment of France to the digitization of its network. The French PT&T was the first to commit its future to digital switching and transmission. France's TRANSPAC data network is in operation and is interconnected to Euronet [14].

In Great Britain, the Post Office is planning an ISDN, centered around System X and a nationwide lightwave network. West Germany, Japan, Italy, and Sweden are all proceeding towards digital switching and transmission. National data networks, which are interconnected into Euronet, now exist in the major European countries. Videotex and Teletex are also emerging in Europe, Japan, Canada, and the U.S. [14].

This quick and incomplete summary of the status of the ISDN evolution around the world gives the flavor of where we are at present. The accomplishments which have been mentioned involve the production and deployment of some parts of the ISDN. Each of these has been deployed largely because it was the economic choice for a specific application, such as a digital local switch, a digital carrier system, or even a digital service network. The combined capital and operational savings available through digital switching and transmission should continue to expand the deployment of digital parts even without an ISDN [14]. See Appendix A for additional details on evolution to an ISDN.

2.2.6. ISDN Applicability to the NASA Space Station

As indicated earlier, the goal of the ISDN is to provide a versatile, multiservice network with standard customer interfaces. ISDN will continue to evolve from the emerging public digital network, and thus should be fairly well defined, and also developed by the 1990's. The fact that most authorities in the communications and computer fields expect the ISDN to come into its own between now and 1990 is of particular significance to planners of the NASA Space Station.

It now is clear that an ISDN type of structure aboard the Space Station is advantageous for several reasons. First, the evolving digital technology upon which the ISDN is based will be well proven, and accepted by the time the Space Station is to become operational. This allows NASA to take advantage of existing state of the art technology. Secondly, the multiservice, versatile standard interface requirements of the ISDN type network will accommodate the voice, data, and video needs of the station quite well. This is especially true if predicted advances in fiber optical technology come to pass allowing for (perceived) unlimited channel capacity. The station's requirements for quick responsiveness and fast turnaround on critical computer computations and processing, dictate very high channel capacity. Also, by designing and installing an ISDN type communications network, the station will not have to expend funds and manhours in determining how best to interface the Space Station with the worldwide ISDN should a clear need to do so exist. The technology will already exist and thus NASA can concentrate on how best to implement the technology so as to provide most efficient and effective service to the station. And finally, the anticipated size of the station is such that an integration of hardware and software functions is essential if all needs are to be met satisfactorily within the 150 to 200 feet area. The concept of an ISDN allows for the most efficient use of a very small work area.

In summary, the massive amounts of video, voice and bulk data requirements of the station can best be met by installing a fast (which implies fiber optics technology), highly modularized (which implies ease of expansion), integrated communications network. These requirements are precisely what the ISDN is all about whether viewed in the context of a local or long-haul network.

2.3. Why spiral Over Traditional Topologies?

Our spiral topology is flexible in that it is easily expanded to any desired number of modules. Although nodes are added four at a time (as a full module), they need not be activated locally until needed; the network still functions properly. The expansion of a spiral configuration requires no restructuring of a complex routing scheme, since there is none. There is no need for each node to maintain routing tables or connectivity matrices for the entire, or partial network. Messages move automatically from node to node based on the spiral and direction flags initialized at the originating node. Also, node failures have minimal impact on operational nodes since messages are automatically routed around failed nodes by the directly connected operational node. Further, with the spiral topology, severe network degradation due to multiple node and module failures does not preclude good nodes from communicating with each other. As long as a path exists between any two nodes, no matter the length, the self-routing algorithm will seek out this path and deliver the message to its destination.

Finally, one major drawback to using fiber optics in local area networks is the relatively high cost (in time delay and money) associated with optical to electrical and electrical to optical conversion of messages for intermediate storage. In particular, the high speed of fiber communication in store and forward environments is somewhat negated by the slower process of converting the message for storage while either a routing decision is made, or the message awaits the outgoing transmission link. Conversion to electrical format for temporary storage is necessary due to a lack of cost effective optical memory technology for message storage. With spiral, since messages are routed on-the-fly, there will be no need to store messages while a routing decision is being made. Only busy links will cause the message to be stored. Thus, the balance between message and circuit switching should be more easily attained. This feature alone opens numerous new opportunities for fiber optics use in local area network environments, and other environments where storing and forwarding of messages is an integral part of transmission.

3. REVIEW OF TRADITIONAL TOPOLOGIES

3.1. Introduction

Four traditional computer network topologies have emerged over the years to form the basis for network structures: 1) bus; 2) ring; 3) star; and 4) mesh. While each has proven advantages for the environments for which they were designed, each also has distinct disadvantages centered around type and volume of traffic, delivery speed requirements, number of users (connected nodes), and geographical placement of these users. Further, given that one of these topologies is selected for a new network, users are then expected to use one of the few access schemes germane to that topology in order to increase chances of the network performing as originally intended.

3.2. Bus Topology

The bus topology (figure 3-1) is good for bursty type traffic whose messages are relatively short. Although this scheme can serve a large number of users, conflicts resulting from attempts to transmit simultaneously by two or more users (called collisions), are inevitable. The probability of collisions, and taking positive steps to minimize them, results in a fairly low theoretical maximum message throughput threshold. Access to a bus topology is usually via some contention scheme, where users compete with each other for access to the transmission channel.

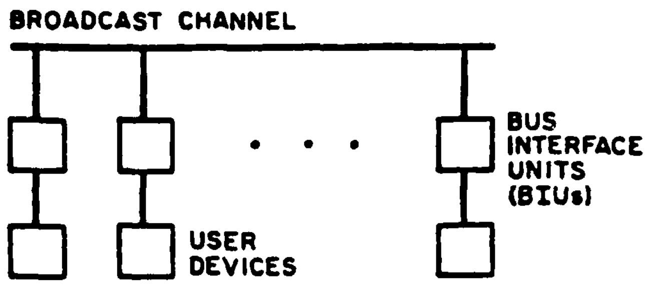


Figure 3-1. Broadcast Bus Topology.

3.3. Ring Topology

In ring topologies (figure 3-2), all users are connected in a circular fashion, and a message is passed around the ring from one user to the next, until its final destination node is reached. There is a maximum, theoretical limit to the number of bits that can appear on a ring at any one time, and throughput and delay are both heavily influenced by the ring size. Further, some central controller usually manages the ring to insure proper operation. In this environment, contention, reservation, or token passing are typical schemes that determine who gets to send next.

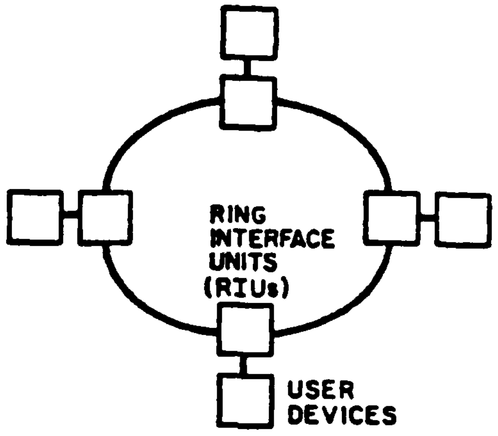


Figure 3-2. Ring Topology.

3.4. Star Topology

The star (figure 3-3) topology relies heavily on a central controller that acts as a "master" to the connected "slave" nodes. Usually polling is the access scheme used here. The major drawback to the star topology is that of failure of the master controller. If the master fails, then the slaves become isolated orphans. Although expanding a star is a bit more straight forward than expanding a ring, every added node means increased average network delay for each, since a greater number of nodes compete for attention.

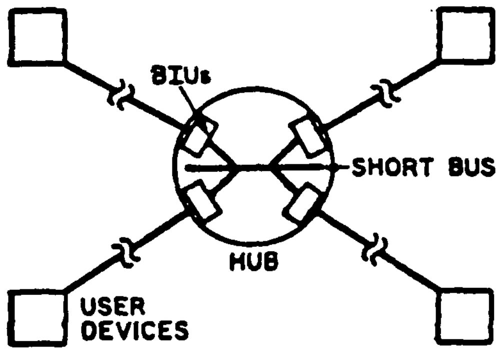


Figure 3-3. Star Topology.

3.5. Fully Connected Mesh Topology

Finally, the fully connected mesh (figure 3-4) topology is the most flexible of the traditional topologies since nodes typically operate at the same precedence level, and failed nodes can be bypassed via alternate routing. Thus, the mesh offers the best opportunity over the other three to reduce network delay and increase throughput for various numbers of nodes. However, if more than a handful of nodes are fully connected, complex network routing and management algorithms are needed for successful operation. Also, most mesh type networks operate in a store and forward mode, where an entire message is received and stored at an intermediate node before it is forwarded to the next node in the message's route path. Therefore, the inherent increased cost in dollars and complexity over the other three traditional topologies, renders the mesh unacceptable to several users who otherwise prefer, and may indeed demand, the speed and flexibility of the mesh. This is especially true in local area network environments.

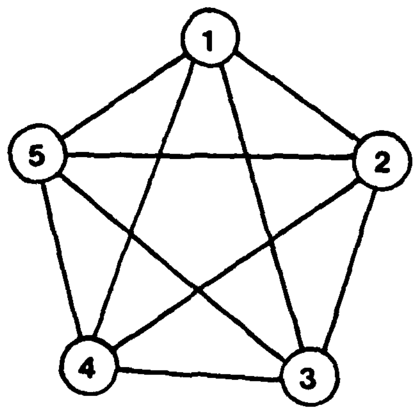


Figure 3-4. Fully Connected Mesh Topology.

4. TIME DELAY FORMULAS FOR LAN TOPOLOGIES

4.1. Introduction

This chapter consolidates the delay formulas for the most popular access protocols used in a Local Area Network (LAN) environment. Contained herein are discussions and comparative results for Carrier Sense Multiple Access with Collision Detection (CSMA/CD), Token Bus and Ring, Slotted Ring, and Ordered Access Bus protocols. These protocols are used in one of the network topologies of figures 3-1 to 3-4.

Computation and analysis of the variance of the waiting time for token rings is found in [22]. Readers interested in the end-to-end performance modeling of LAN's are referred to [23] and [24].

4.2. Carrier Sense Multiple-Access with Collision Detection (CSMA/CD)

4.2.1. Introduction

The delay formula for CSMA/CD presented in this section is that developed by Werner Bux [25], and altered algebraically by Schwartz [26]. It is very close to Simon Lam's result [27]. The primary difference between the Bux and Lam delay formulas, and those developed by Leonard Kleinrock and Fouad Tobagi in [28] is as follows. Bux and Lam assume that in the local networking environment, collisions in the channel are detected and that users involved in a collision abort their transmissions immediately upon detection. Mechanisms for detecting collisions and aborting collided transmissions have been implemented in several multipoint cable networks [29, 30, 31]. However, it appears to be much more difficult to implement a "collision abort" capability in the radio environment of Kleinrock and Tobagi's work [27].

4.2.2. Model Assumptions

Like the p-persistent protocol in [28], local network users are assumed to be time synchronized so that following each successful transmission, the channel is slotted in time. Users can start transmission only at the beginning of a time slot. τ represents the channel propagation delay, and the minimum slot size is 2τ. To enhance the validity of his comparisons of various access protocols in a continuous, nonslotted environment, Bux heuristically modifies Lam's formula in [27] by reducing the mean delay by τ [25]. Figure 4-1 illustrates Bux's CSMA/CD scheme. His result presented here defines the mean transfer time T as the queueing and access delay at the sender, the transmission time of the packet, and the propagation delay. The protocol is defined by two possible courses of action for ready users:

1). Following a successful transmission, each ready user transmits with probability 1.

2). Upon detection of a collision, each ready user uses an adaptive algorithm for selecting its retransmission probability during the next slot.

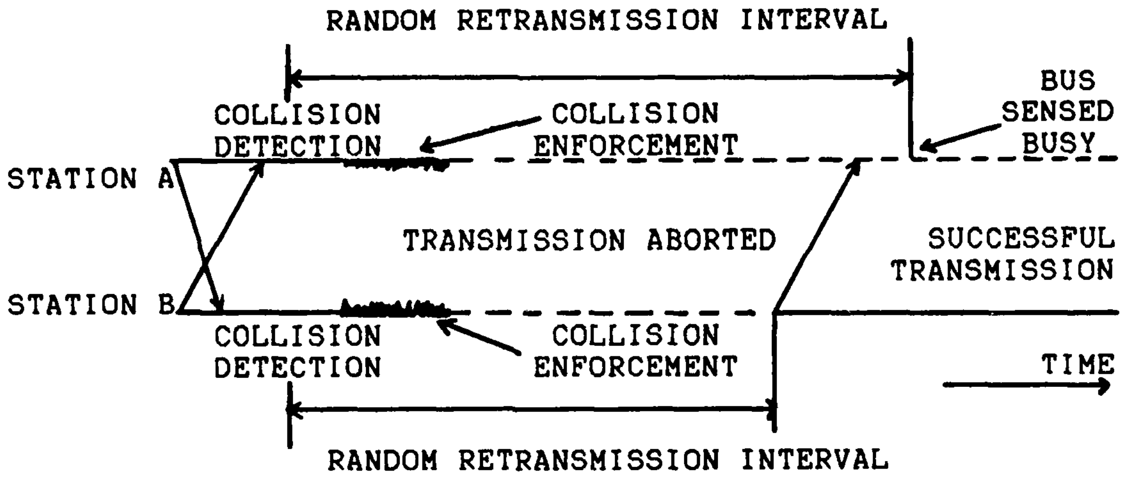


Figure 4-1. CSMA/CD Bus: Example of Operation.

The model assumes errors due to random noise are insignificant relative to errors due to collision, and can be neglected. The traffic source is an infinite population of users who collectively form an independent Poisson process with an aggregate mean message generation rate of λ messages per second. It also assumes transmission times of each message is an independently distributed random variable. Comparison of delay versus throughput results are shown in figure 4-2 . The delay formula is:

$$T = \frac{\lambda\{E[T_p{}^2] + (4e + 2)\tau E[T_p] + 5\tau^2 + 4e(2e - 1)\tau^2\}}{2\{1 - \lambda(E[T_p] + \tau + 2e\tau)\}}$$

$$+ E[T_p] + 2\tau e + \frac{\tau}{2}$$

$$- \frac{(1 - e^{-2\lambda\tau})(2/\lambda + 2\tau/e - 6\tau)}{2\{F_p{}^*(\lambda)e^{-\lambda\tau}1/e - 1 + e^{-2\lambda\tau}\}}$$

where:

$E[T_p]$	=	mean service time,
$E[T_p{}^2]$	=	second moment of service time,
λ	=	aggregate arrival rate,
τ	=	propagation delay, and
$F_p{}^*(\lambda)$	=	Laplace Transform of the probability density for Tp.

The fourth candidate for local area subnet use presented by Bux is also included in figure 4-2 for comparison. Section

4.6 contains a comparable discussion of this ordered access scheme (multilevel multiple-access (MLMA)), and additional details can be found in [25].

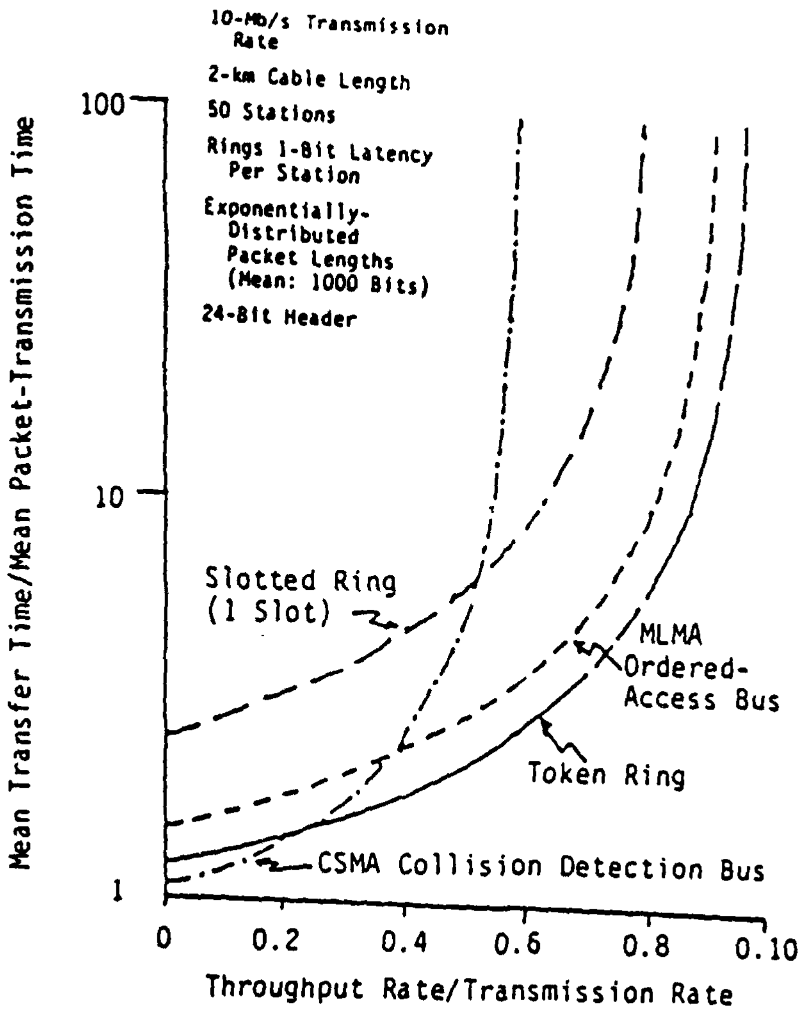


Figure 4-2. Delay-Throughput Performance of Various Protocols.

4.3. Token Ring

4.3.1. Introduction

The formula presented in this section is also due to Bux [25]. His result is similar to formula (1), page 132 of the work by DeMoraes and Rubin [32].

4.3.2. Model Assumptions

In a token ring, access to the transmission channel is controlled by passing a permission token around the ring. The model assumes a population of S terminals. Messages arrive at terminals in accordance with a Poisson process with aggregate rate λ. Here τ represents the round-trip delay in the ring, including possible delays of the signals caused within each station. In the comparative results, τ was assumed to be 5 microsec per kilometer of cable length.

With respect to the order of service, two unique policies are cited by Bux: 1) A queue is served until it is empty ("exhaustive service"); and 2) Only a limited number of packets (e.g. one) is served per access possibility ("non-exhaustive service"). Although, in principle, performance differences exist-between these policies [33], Bux contends that in a local network, these differences are small if traffic is uniformly distributed among stations. In the model for which the formula applies, a new free token was generated immediately after the last bit of a packet had left the source. This implies the possibility for multiple tokens on the ring. At most one of them however, is in the free state. From reliability and recovery points of view, it may be desirable to have no more than one token at a time on the ring. This can be done in two ways [25]. 1) The sender issues a new token after he has completely removed his entire packet ("single-packet" operation). In Bux's model, this can be taken into account by prolonging the packet service time T_p by the total ring round-trip delay τ. 2) A more efficient solution is that the sender does not issue a new token before he has received his own token back ("single-token" operation). This rule becomes effective in cases when a packet is shorter than the ring latency. In the performance model, this can be described by setting T_p equal to τ for all packets shorter than the ring latency. Comparative results are presented in figure 4-2, and the formula for the mean delay is:

$$T = \frac{\rho E[T_p{}^2]}{2(1 - \rho)E[T_p]} + E[T_p] + \frac{\tau(1 - \rho/S)}{2(1 - \rho)} + \frac{\tau}{2}$$

where:

S = number of connected stations and

$\rho = \lambda E[T_p]$.

4.4. Token Bus

In this access scheme, the token controls access to the shared bus. As in the token ring, the terminal holding the token has momentary control of the medium. In the bus configuration, however, the terminals are connected parallel to the medium. Thus once a terminal transmits the token, its signal is received by all terminals. Unlike the ring, for token bus operation, terminal (i) does not have to be physically adjacent to terminal (i+1) in order to transmit in that sequence. Hence, a token bus behaves like a logical ring.

Performance results cited above for token ring are appropriate for the token bus case, if it operates as a logical ring. However, now the propagation delay represents the total latency for a round-trip around the logical ring. This value may be larger than for the token ring since we now make the delay from one end of the bus to the other, (worst case) plus the time for a terminal interface device to process the signal. DeMoraes and Rubin's formulas (1) and (12) in [32] for token ring and token bus respectively, are identical to Bux's result except for notation. Figures 4-2, 4-3, and 4-4 contain comparative results for token bus and token ring. The result for a token bus that behaves like a logical ring is repeated from the previous section. That delay formula is:

$$T = \frac{\rho E[T_p]}{2(1 - \rho)E[T_p]} + E[T_p] + \frac{(1 - \rho/S)}{2(1 - \rho)} + \frac{\tau}{2}$$

where:

S = number of connected stations,

$\rho = \lambda E[T_p]$, and

τ = round-trip propagation delay.

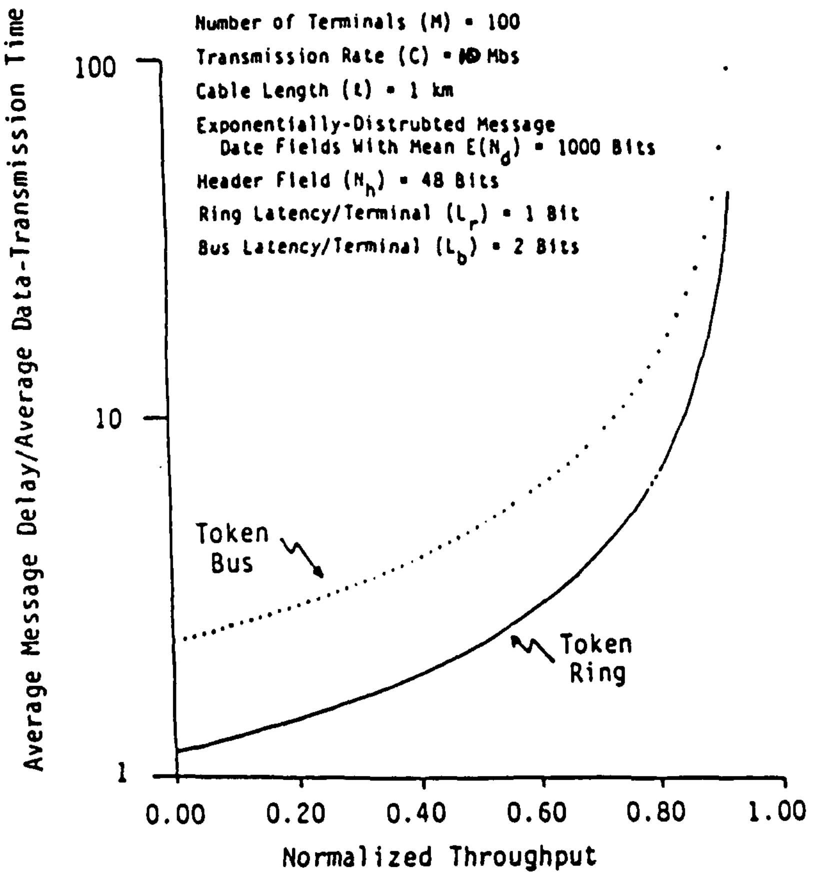


Figure 4-3. Average Message Delay vs. Throughput for Token Bus and Token Ring Under Balanced Traffic ($\lambda_i = \lambda$; i = 1,2,3,...M) and Gated Service (10Mbps).

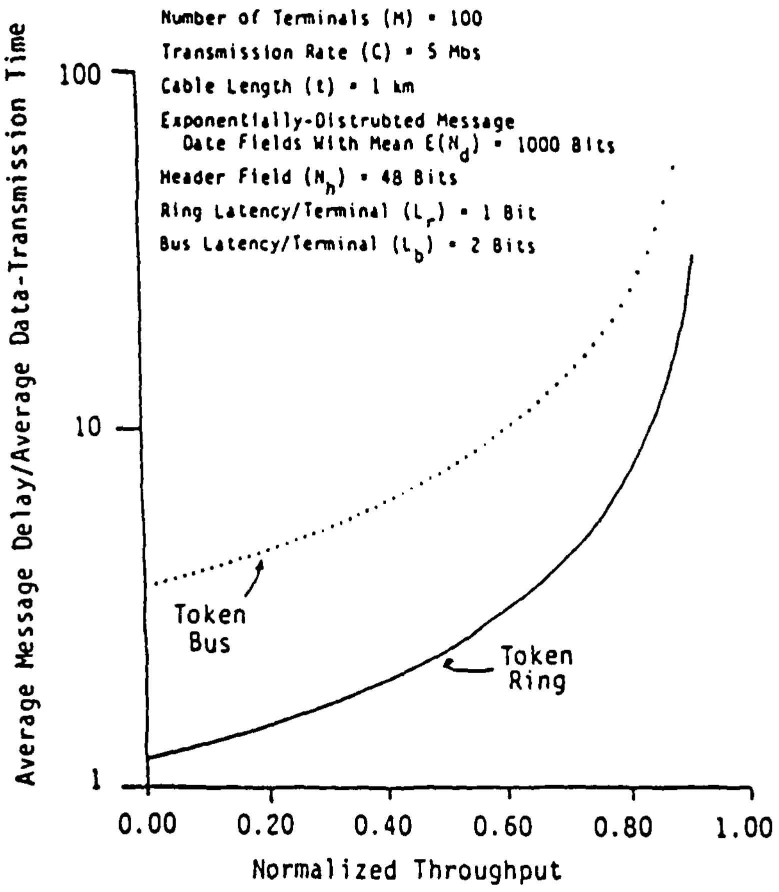


Figure 4-4. Average Message Delay vs. Throughput for Token Bus and Token Ring Under Balanced Traffic and Gated Service (5Mbps).

4.5. Slotted Ring

4.5.1. Introduction

In the type of slotted ring studied by Bux [25], a constant number of fixed-length slots continuously circulate around the ring. A full/empty indicator in the slot header is used to signal the state of a slot. Any ready station occupies the first empty slot by setting the full/empty indicator to "full", and places its data into the slot. When the sender receives back the occupied slot, it changes the indicator back to "empty". Several slotted ring models are developed and evaluated in [24].

4.5.2. Model Assumptions

Usually, the slots in local rings are short, which means that a packet or message has to be transmitted using several slots. Packets queued at a station are served in sequence for a short time quantum Δt, which corresponds to the slot length. On the average, packets are usually at least ten times longer that the slot. Bux further adequately justifies in [25] key assumptions that led to the results presented here. A most critical one is that the ring bandwidth can be fully utilized. This means he assumes for simplicity that the following relation holds among ring latency τ, transmission rate v, slot length L_h and L_d

(length of header and data), and the number of slots σ:

$$\tau v = \sigma(L_h + L_d).$$

Thus Bux assumes that the ring latency·times transmission rate equals number of slots times slot length.

Comparative results are in figure 4-2. The simple expression for the mean delay is:

$$T = \frac{2}{1 - \rho} E[T_p] + \frac{\tau}{2}.$$

4.6. Ordered Access Bus

4.6.1. Introduction

The model described here for the ordered access bus approach is again due to Bux [25], and applicable in a star configuration since the star is known to have a short, fast bus to which users either contend for, or are polled to gain access. Information transmission occurs in variable-length frames. A controller provides start flags at appropriate time intervals which signal the beginning of a frame. The frame is divided into a request slot and an arbitrary number of packets. In the version of multilevel multiple-access (MLMA) analyzed in [25], every station attached to the bus owns one bit within the request slot. By setting its private bit, a station indicates that it wants to transmit a packet within this frame. At the end of the request cycle, all stations know which of the stations will make use of this frame. The transmission sequence is given by a priority assignment known to all stations.

4.6.2. Model Assumptions

The bus is modeled as a single-server facility. Newly generated packets arriving during the current frame must wait until the new frame starts for transmission. To ensure that all stations know the entries made in the request slot, the scheduling time T_S may have to be significantly longer than the pure transmission time of the request slot. In Bux's version, T_S equals twice the time needed to transmit S bits plus the propagation delay τ. The underlying assumption of his model is that the distance between two stations transmitting in succession is uniformly distributed between zero and the maximum bus length. The appendix of [25] contains the derivation of the mean transfer time T, and comparative results are at figure 4-2. The delay formula is:

$$T = \frac{\rho'(E[T_p^2] + \tau E[T_p] + \tau^2/3)}{2(1 - \rho')(E[T_p] + \tau/2)} + E[T_p] + \frac{\tau}{2}$$

$$+ \frac{(3 - \rho')}{(1 - \rho')} \frac{T_S}{2}$$

where:

$\rho' = \lambda\{E[T_p] + \tau/2\}$, and

$T_s =$ mean scheduling time.

4.7. Comments on the Star Configuration Access Methods

The star configuration may be implemented using various access schemes. The fact that the star configuration exists in a network may be more of a consequence of the access protocol rather than explicit star design. For example, the classic polling of attached terminals may be accomplished in a logical "star" arrangement with the host/polling computer as the hub. Terminal response times and delay formulas for polling may be found in [34] and [35], for example.

As indicated earlier, the ordered access configuration MLMA may be applied to the short-bus (star) topology of figure 3-3. Delay calculations can be done by using the equation presented in section 4.6, for the Ordered Access Bus.

A detailed study of the star topology addressing different access protocols with emphasis on network performance, was conducted by Kamal and reported in [36].

Finally, Anthony Acampora, C. D. Tsao, and M. Hluchyj in [37] and [38] discuss a new local area network using a centralized bus. Although explicit delay formulations were not included in their articles, the intent of their technology (based on DATAKIT packet switching [39]) is to combine the advantages of bus, ring and star architectures, avoiding the disadvantages of each.

4.8. Effect of Propagation Delay and Transmission Rate on Performance

In analyzing local network performance, according to William Stallings [40], the two most useful parameters are the data rate (R), of the medium, and the average signal propagation delay (D), between stations on the network. In fact, it is the product R x D that is the single most important parameter for determining the performance of a local network. (The data rate times the delay product equals the length of the transmission medium in bits.) The length of the medium in bits compared to the length of the typical packet is usually denoted by a:

$$a = (R \times D)/L = \text{Propagation time}/\text{Transmission time}.$$

The maximum possible utilization of a network can be expressed as the ratio of total throughput of the system to the capacity or the bandwidth:

$$\begin{aligned} U &= \text{throughput}/R \\ &= (L/(\text{propagation} + \text{transmission time}))/R \\ &= (L/(D + L/R))/R \\ &= 1/(1 + a) \end{aligned}$$

Thus we can clearly see that **a** determines an upper bound on the utilization of a local network, regardless of the medium access protocol used. In particular, Stallings' throughput results for various protocol access methods are summarized below as [40]:

Token_Ring_and_Token_Bus

$S = 1/\{1 + a/N\}$, $\quad a < 1$

$S = 1/\{a(1 + 1/N)\}$, $\quad a > 1$

N = number of stations.

CSMA/CD

$S = 1/\{1 + 2a(1 - A)/A\}$,

A = Probability that exactly one station transmits in a slot.

So in terms of network performance at high speeds (such as those possible via optical fiber technology), the limiting value is reached as a result of adjusting to ensure the acceptable value of **a**. Since $a = RxD/L$, to keep it constant, if the rate is to increase, the propagation delay D must be decreased (shorten the medium or use a "better" physical medium), or the packet length L must increase. Increasing L could lead to greater inefficiencies if doing so results in a large number of partially filled packets [40].

Figure 4-5 depicts the relationship of throughput versus offered load as a function of **a**, independent of access protocol. In [41], Bart Stucks develops expressions for the **maximum** mean throughput rates for various LAN access schemes. Figure 4-6 compares delay versus throughput for polling and CSMA/CD given various values of **a** [27]. Figure 4-7 shows normalized throughput as a function of **a** for various numbers of stations (N). And finally, figure 4-8 shows simulation results for maximum potential data rates for the most popular LAN protocols [40].

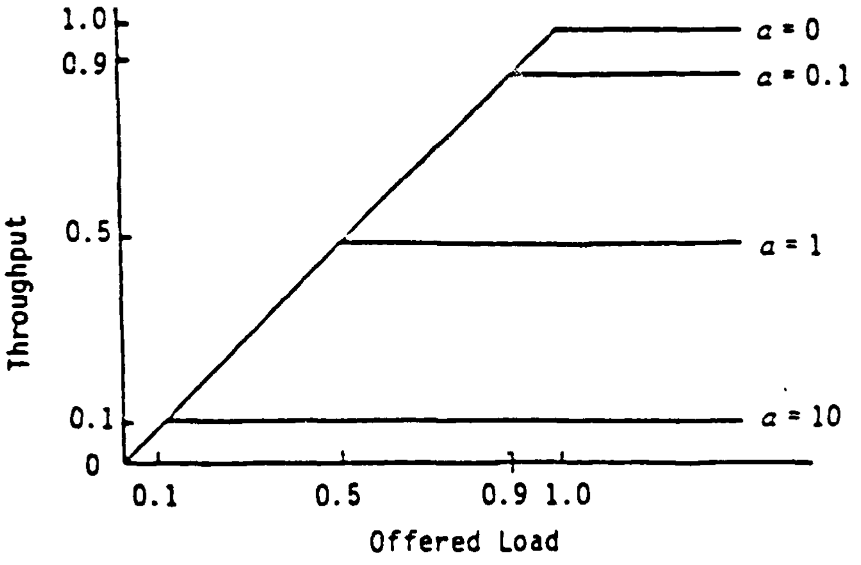


Figure 4-5. Throughput vs. Offered Load.

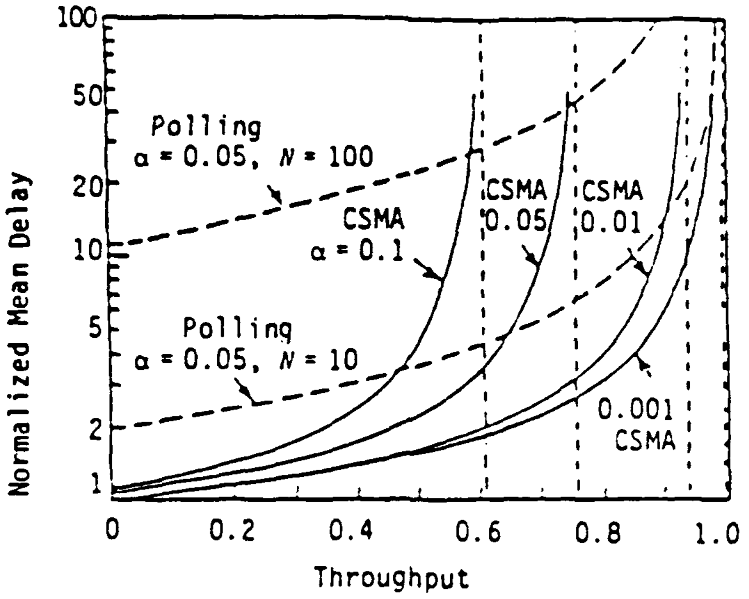


Figure 4-6. Mean Delay vs. Throughput: Polling and CSMA/CD.

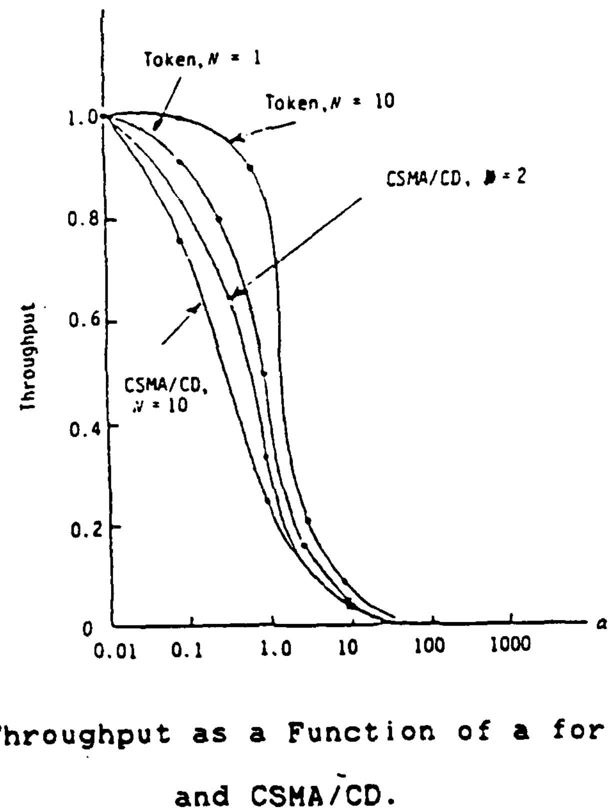
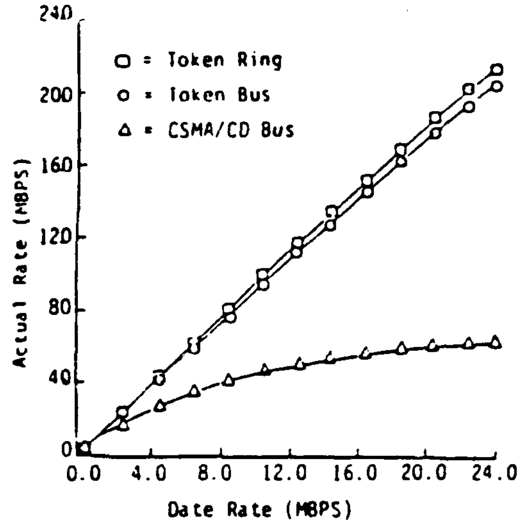
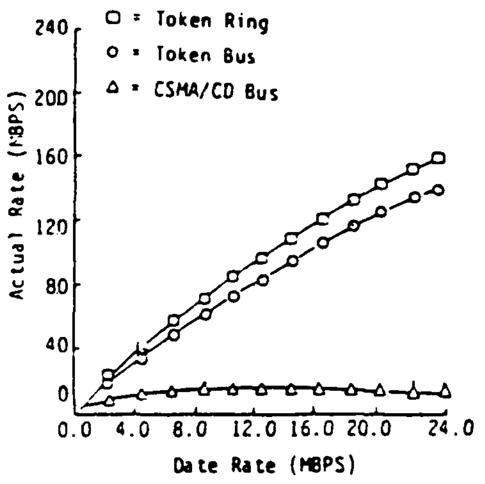


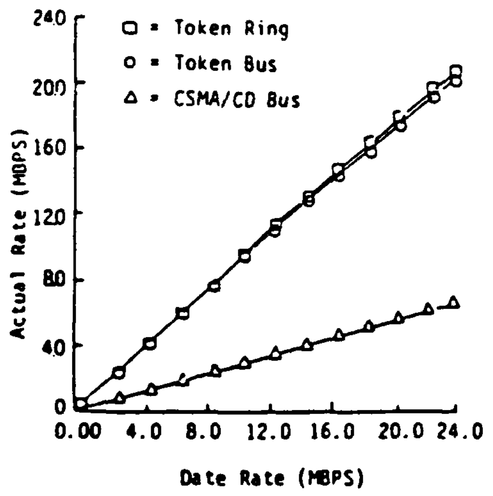
Figure 4-7. Throughput as a Function of a for Token Passing and CSMA/CD.



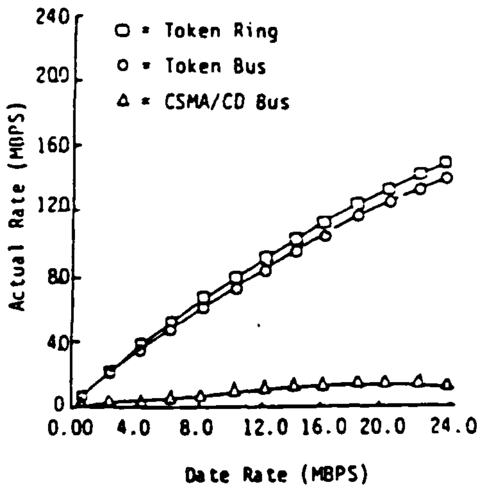
(A) 2000 Bits Per Packet, 100 Stations Active Out of 100 Stations Total



(B) 500 Bits Per Packet, 100 Stations Active Out of 100 Stations Total



(C) 2000 Bits Per Packet, 1 Station Active Out of 100 Stations Total



(D) 500 Bits Per Packet, 1 Station Active Out of 100 Stations Total

Figure 4-8. Maximum Potential Data Rate for LAN Protocols.

4.9. General Results for Other Network Approaches

Other more general approaches to local area network implementations are summarized below. These include packet switching using partially connected mesh topologies, circuit switching, and frequency division multiplexed (FDM) switching, or its optical equivalent, wave division multiplexed (WDM) switching.

Packet Switching on Partially Connected Mesh Networks:

$$T = \sum_{i=1}^{M} \frac{\lambda_i}{\gamma} \frac{E(T_p)}{1 - \lambda_i E(T_p)}$$

where:

λ_i = flow on link "i", and

γ = total offered traffic.

Circuit Switching:

$T = E(T_p)$ + Time Awaiting Permission.

Fast Packet Switching:

$T = E(T_p)$ + Intermediate Node Switching Delay.

TDM Switching:

Basically, the same as slotted ring.

FDM Switching:

Fixed allocated resources.

5. SPIRAL NETWORK DESCRIPTION

5.1. Architecture

The minimum spiral network contains four modules of four computer nodes each for a total of 16 nodes. A module is formed by fully connecting four nodes. An internal base 4 numbering scheme is used throughout the spiral network, regardless of size. These internal base four numbers are the basis upon which the self-routing algorithm is built, and therefore, form the heart of our fault tolerance strategy.

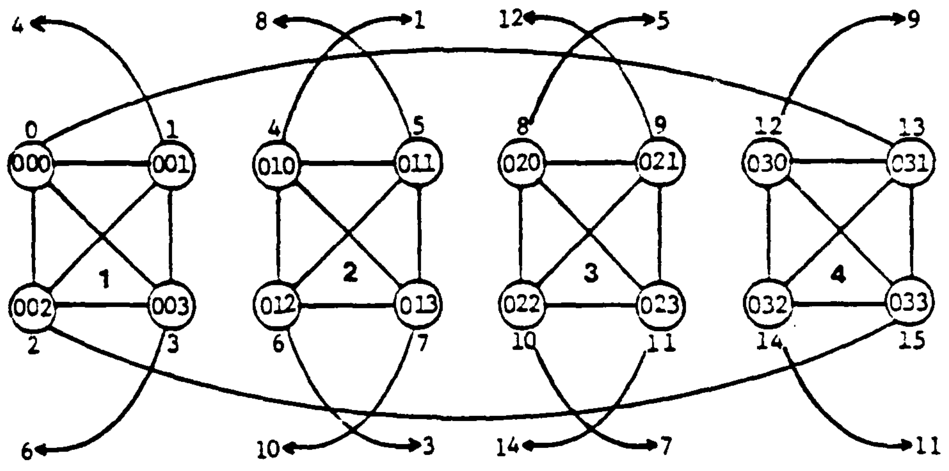


Figure 5-1. Minimum Four Module Spiral Network.

Figure 5-1 shows the minimum four module, 16 node spiral network. Actually, the minimum four module spiral network is better displayed in figure 5-2, except in that configuration, the six pair of links used to expand the network to any desired size are not obvious. Spiral is expanded by first fully connecting, in groups of four, the nodes to be added. These nodes are assigned the next consecutive base 4 numbers available beyond the highest base 4 numbered node in the current spiral network. Then this new module is brought into the existing topological structure by altering the six pair of links shown in figure 5-1. Figure 5-3 shows the result of adding a single module to the minimum four module topology of figure 5-1. Figures

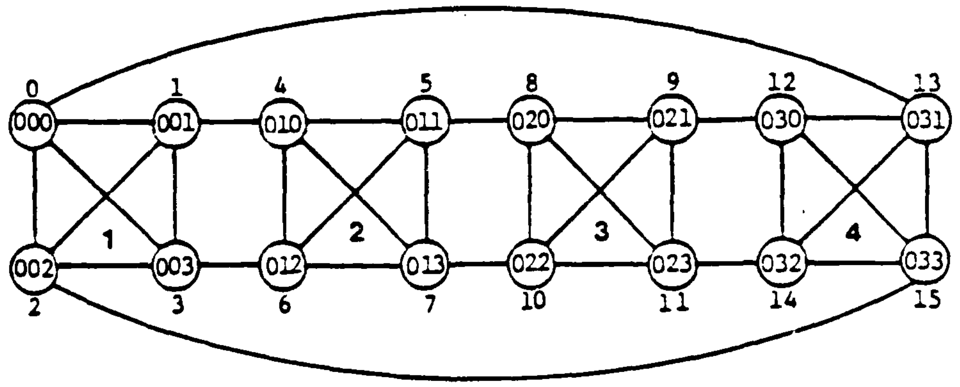


Figure 5-2. Alternate Form of Minimum Spiral Network.

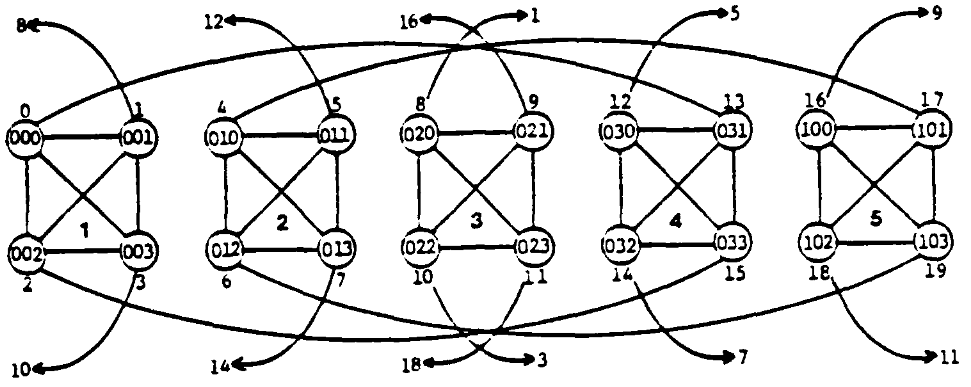
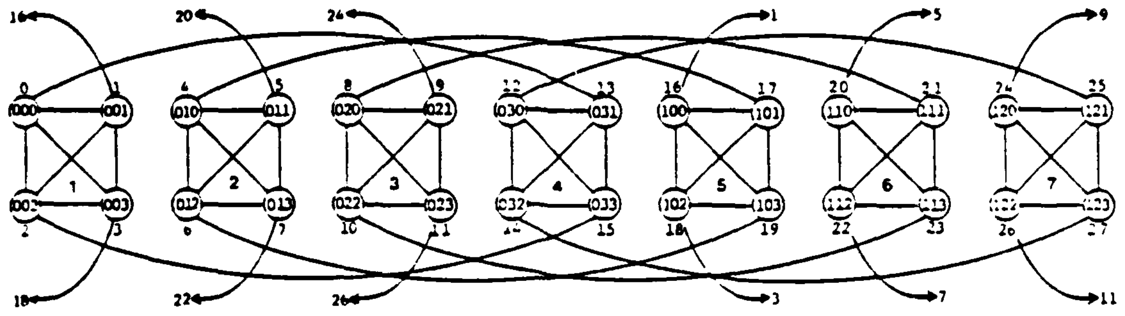
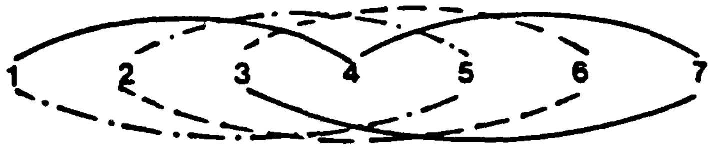


Figure 5-3. Five Module Spiral Network.

5-4 through 5-8 depict 7, 8, 11, 13, and 14 module spiral networks. Notice in each case the six pair of links resulting from the expansion. Clearly, as more modules are added, the ease in expanding the network becomes more evident since the six pair of links become more pronounced. Close examination of figures 5-7 and 5-8 (the 13 and 14 module cases) reveals that as the spiral network grows, there is a pool of modules totally unaffected by the addition of new modules. The significance in this point is that the network can be expanded quite easily by temporarily disabling nodes directly attached to the six pair of links.

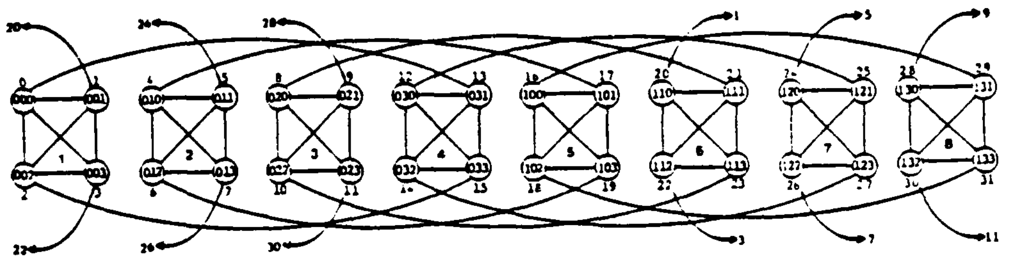


a). Seven Module Spiral Network.

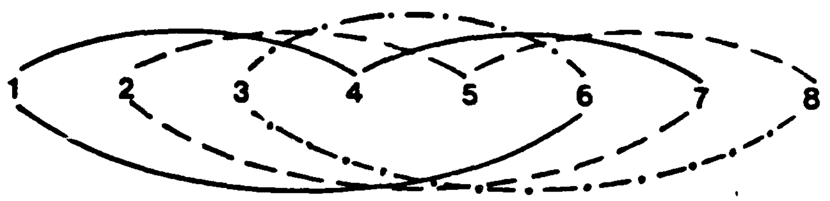


b). Threading Pattern.

Figure 5-4. Seven Module Spiral Network and Threading Pattern.



a). Eight Module Spiral Network.



b). Threading Pattern.

Figure 5-5. Eight Module Spiral Network and Threading Pattern.

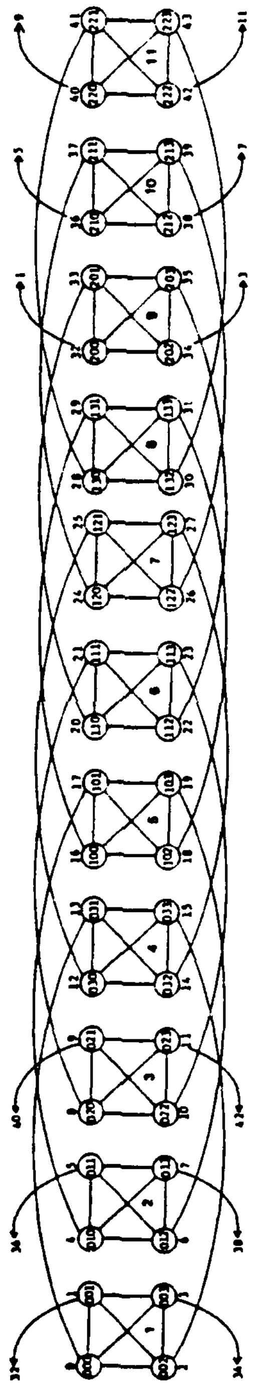


Figure 5-6. Eleven Module Spiral Network.

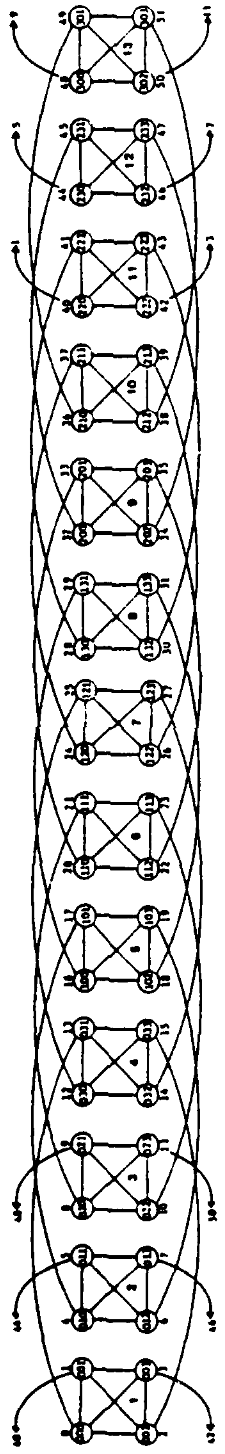


Figure 5-7. Thirteen Module Spiral Network.

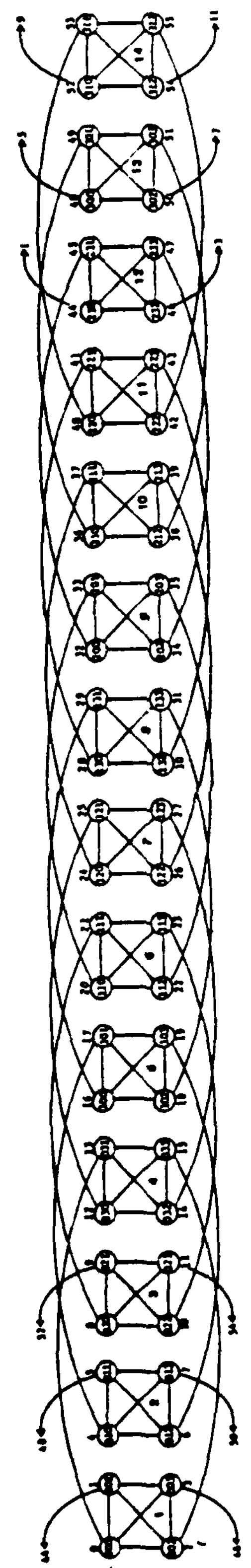
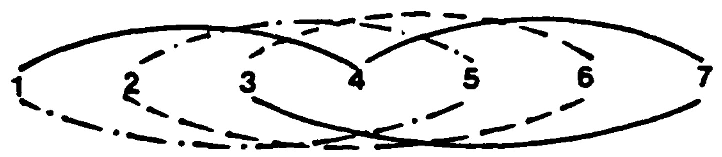


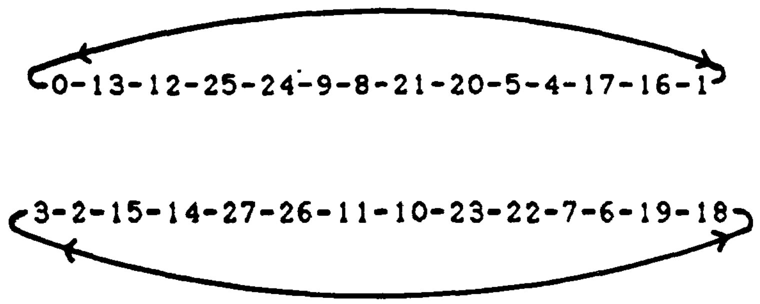
Figure 5-8. Fourteen Module Spiral Network.

All other nodes continue operation as usual even as the topology is being expanded. This last point is one of the major attributes of the spiral topology. When expanding any of the four traditional topologies, not only is the overall network affected during the expansion, routing and theoretical limits must also be altered. These alterations are usually at the expense of temporary inoperability of the network.

In the spiral connectivity scheme, every third module is threaded together top and bottom in a circular fashion. The process of bringing new modules into an existing topology is akin to adding an element into a doubly linked circular list. Both nodes on the front end of each module are connected to the nodes on the rear side of its link module. This threading pattern continues until the last pair of front links loops back around and connects with the rear side of the front end module. Figures 5-9 and 5-10 repeat the network threading patterns for the seven module (28 node) and eight module (32 node) spiral networks. These threading patterns reflect the order in which modules are encountered as a message proceeds towards its destination. The threading pattern is also a quick check that all modules are reachable.



a). Seven Module Threading Pattern.



b). Unwound Spirals.

Figure 5-9. Module Threading Pattern and Spirals.

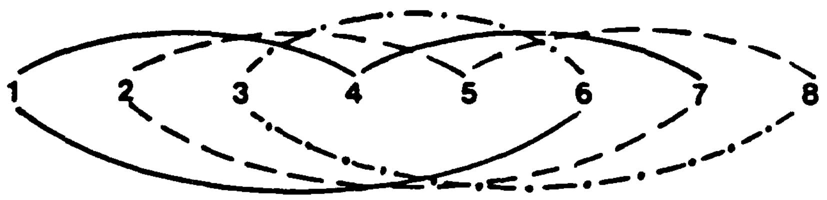


Figure 5-10. Eight Module Network Threading Pattern.

Figure 5-9(b) shows the two unwound top and bottom spirals for the seven module network. Evident here are the four totally disjoint (nonoverlapping) paths that exist between any source-destination pair. In any size spiral network that has no failed nodes or links, messages can travel along the top spiral right, top spiral left, bottom spiral right, or bottom spiral left. The Preprocessing Algorithm at the source node determines which of these four routes is the shortest path to destination. Spiral and direction flags (SF and DF) are then initialized, and used by the self-routing algorithm for on-the-fly routing from source to destination.

The aforementioned architecture's connectivity scheme remains fixed, no matter the desired size of the resulting spiral network. Since every third module is threaded together to form the desired spiral network, the total number of resulting modules must not be an integer multiple of three. When the spiral connectivity algorithm is applied to modules (m) whose total number is directly divisible by three, the resulting topology is a network that partitions into three subnets of m/3 modules each. This special attribute of the connectivity algorithm leads to a key physical security advantage detailed further in section 5.3 of this chapter.

5.2. Typical Node Configuration

5.2.1. Physical Characteristics

All nodes in a spiral network are the same, although they need not be. The functions performed for error free routing, and routing when failed links or nodes exist, are also identical. None of the nodes is designated the "master" to other "slave" nodes. Further, no node maintains connectivity data for the entire network. The absence of the need to maintain such information, means there is no delay associated with disseminating update status information to the other nodes. Each node maintains status information on the nodes to which it is directly connected only.

Figure 5-11 is a simple graphic representation of a typical spiral topology node.

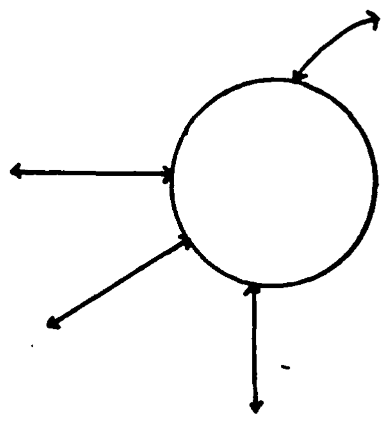


Figure 5-11. Typical Spiral Node.

The nodes are connected by four full duplex links. Three of the links connect to the other three nodes that form the module. The forth connection is the link node that extends to the next (or previous) module, as depicted in the module threading pattern, (See figure 5-9, for example, for the threading pattern of the seven module, 28 node network.) All links operate at a common speed, which can be set as necessary to achieve the desired performance level.

5.2.2. Input Configuration of a Spiral Node

Figure 5-12 represents the message processing procedure followed during a typical message arrival state. Traffic arriving on each of the four links is composed of transient messages on their way towards final destinations, and newly arriving messages to the network.

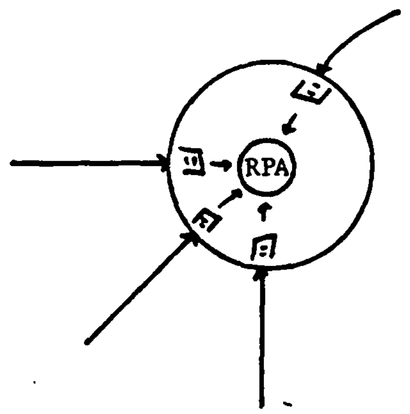


Figure 5-12. Input Configuration of a Spiral Node.

Depending on current busy state of the node, these messages may be queued for processing. Of course if the link speeds are set for extremely fast processing, then these queues should be extremely short. A typical message has either reached its destination module, or must be sent on to the next link module in its path. If the current module is the destination, then the Rapid Processing Algorithm (RPA) gates that message to the final destination node of that module, if the current node is not the final destination node. If the current module is not the final destination, then the Rapid Processing Algorithm gates the message to the appropriate link node. This link node is found by checking the spiral-destination flag pair. Additional routing algorithm details are found in chapter 6.

5.2.3. Output Configuration of a Spiral Node

Figure 5-13 shows the output side of a typical node. Whether a message is transient or a new arrival makes no difference to the Rapid Processing Algorithm. All messages are gated to the appropriate output link from the typical node. The choice of which link to use is governed by whether the current module is the destination module for the message, or only an intermediate module along the message's path. If the Rapid Processing Algorithm determines that one (or more) of the output links/nodes has failed, the spiral or direction flag is complemented. If the failed link/node is the final destination, a message of non-delivery due to failure, can be sent back to the sending node. If the failed link/node is an intermediate one in the path of the message, then the new SFDF pair is used to automatically adjust the path of the message around the failure. See chapter 6 for additional details on routing with and without failures in the network.

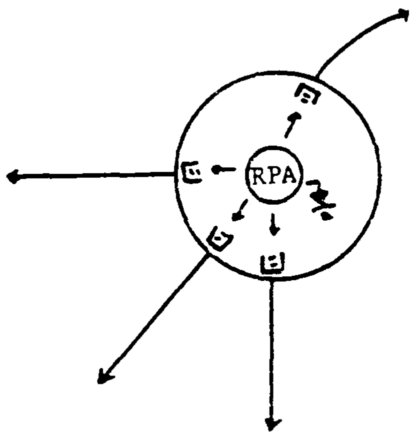


Figure 5-13. Output Configuration of a Spiral Node.

5.3. Key Spiral Network Features

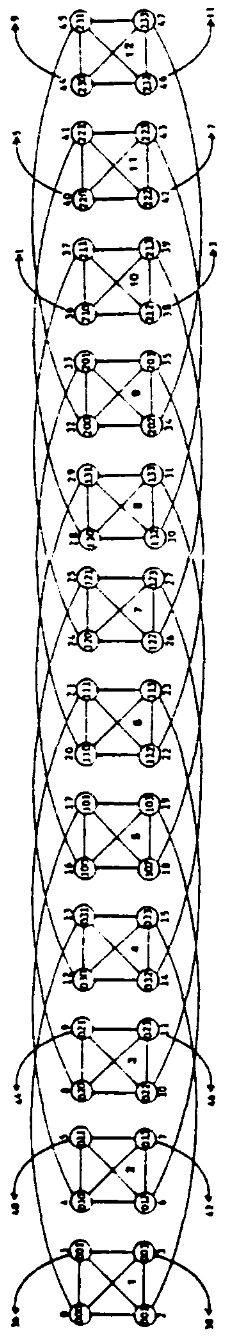
The six most important direct consequences of the spiral topology are 1) ease of expansion; 2) fast, on-the-fly self-routing; 3) extremely high tolerance to faults; 4) increased network security; 5) potential for the total elimination of store and forward transmission due to routing decision delays; and 6) rendering the maximum path length issue moot.

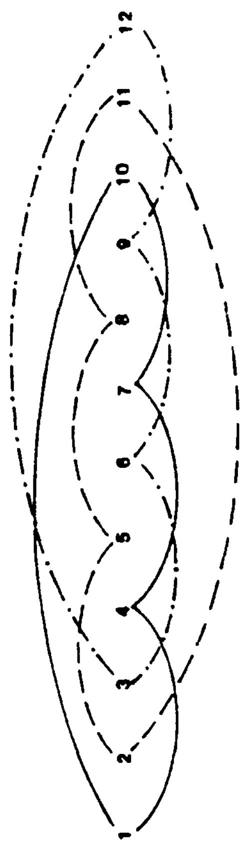
Perhaps the three most significant features are its ease of expansion, fast, on-the-fly self-routing attribute, and spiral's extremely high tolerance to failed links or nodes throughout the network. These two latter features are further amplified upon in chapter 6.

The ease at which the spiral network can be altered, whether to make larger or smaller networks, is evident from the previous description of the spiral architecture. Yet another unique feature of spiral has significant security ramifications.

The construction and operation of the spiral topology is the same regardless of network size, with only one constraint. We contend, however, that this constraint is a major security attribute. Because of our connectivity scheme which directly links every third module, the number of mod· ·es in the spiral topology must not be an integer multiple of three. When the spiral connectivity pattern is applied to modules whose number is a multiple of three, the network partitions into three separate but equal subnets of total_nodes/3 nodes each. There is no possible way for the three subnets to interact. For example, figure 5-14(a) shows a 12 module, 48 node network using the spiral connectivity pattern. In figure 5-15(a) is a 15 module, 60 node network connected using the spiral connectivity algorithm. Upon close examination of the resulting network threading patterns of figures 5-14(b) and 5-15(b), one notices that the 12 modules are completely partitioned into three separate subnets of four modules (16 nodes) each. The 15 modules are partitioned into three subnets of 5 modules (20 nodes) each. That is, we now have the case of the minimum four module (16 node) network of figure 5-1 duplicated three times, and the five module (20 node) network of figure 5-3 duplicated three times. If 21 modules are connected, then the network partitions into three subnets of seven modules (28 nodes) each, and thus duplicates the seven module network of figure 5-4. And herein lies the significant security implication.

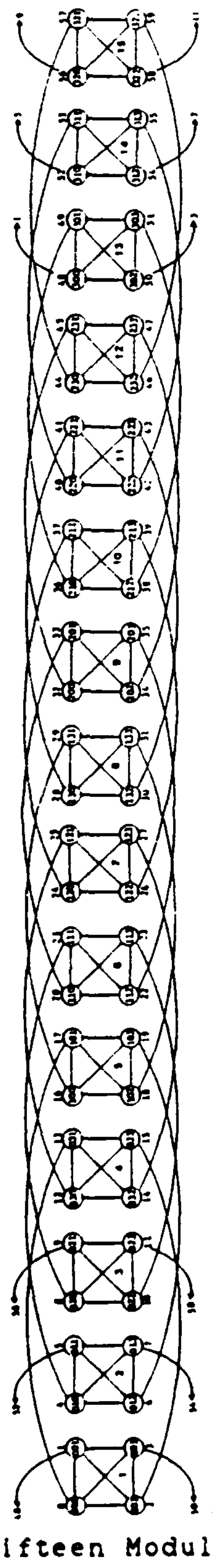
In environments where highly sensitive and classified nodes operate in conjunction with unclassified, routine common user nodes, the ability to temporarily partition the network with ease may be a major network design criterion. If there


a). Twelve Modules.

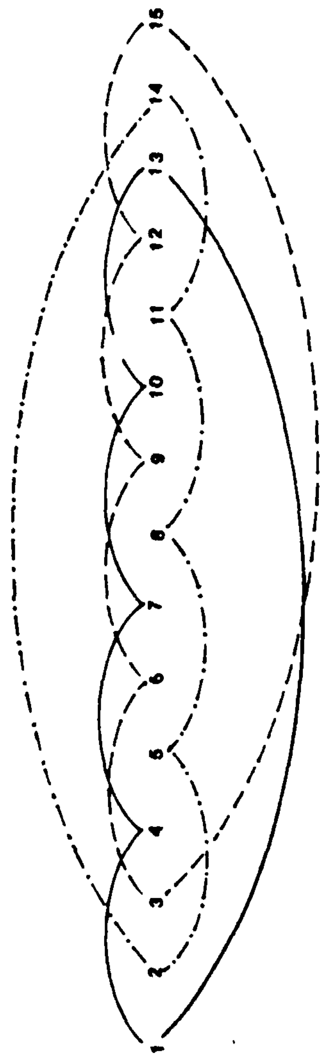


b). Threading Pattern.

Figure 5-14. Twelve Module Network and Threading Pattern.



a). Fifteen Modules.



b). Threading Pattern.

Figure 5-15. Fifteen Module Network and Threading Pattern.

is a recurring need for added, or assured security, whereby only a portion of the total nodes desire to interact among themselves, then the network planners can identify these "super nodes" at network design time. They then can place these nodes in the proper logical positions within a fully connected spiral network. Operation proceeds as normal until the temporary need for added security arises. At that time, by increasing the number of modules to the next closest multiple of three, the network partitions into the three subnets. One of these will be the subnet consisting of the "super nodes" only. The two remaining lower priority subnets still have the ability to communicate with other nodes on their subnet. This temporary network expansion to the nearest multiple of three is simple, and fairly inexpensive. The temporary expansion need not involve the expense of acquiring standby hardware. A single, existing computer can be used to simulate the additional nodes. When the temporary need expires, the network can be returned to its original configuration by deleting the temporary artificial nodes. Recall from section 5.1 of this chapter that the expansion can occur without bringing the network down.. A broadcast message could be sent to each node advising of the impending partitioning into subnets.

The next spiral network feature stems from the use of fiber optic technology and the communications industry's plans to evolve to an Integrated Services Digital Network (chapter

2).

To take advantage of the tremendous speed associated with fiber optics transmission using our spiral topology, the network manager or designer chooses a high enough link speed so that the mean queue length of the spiral network is very small at steady state. While queue buildup is unavoidable, the self-routing algorithm precludes the need to store messages while routing decisions are being make, and sufficiently high link speeds minimize the need to queue the message for transmission on the outgoing link. With current topologies using optical transmissions to transport messages, optical-to-electrical conversion of messages is required before an intermediate message can be stored. These conversions are necessitated by the absence of viable optical memory units. Once a routing decision for the message has been made, the message must then be converted back to its optical format for further transmission. Spiral's on-the-fly routing attribute precludes the need to store messages at intermediate nodes for routing, which means there will be no need to convert the message from optical-to-electrical and vice versa. As expected, in this scenario, average link utilization may be drastically reduced over the store and forward mode of operation. But a lower link utilization is a reasonable tradeoff if speed is indeed vital. Speed is extremely vital for voice and realtime video traffic in an ISDN environment.

Further, by using the spiral topology and its inherent on-the-fly routing attribute, network planners will be better equipped to decide whether or not packet switching is more suitable than circuit switching. In the past, circuit switching was used to ensure uninterrupted transmission of critical, realtime traffic. The path established from source to destination for the circuit was completely dedicated to users for the entire duration of the transmission. With spiral, and the careful selection of link speeds, the overall network throughput and efficiency may be increased by using packet switching, or fast packet switching.

Finally, since the network is self-routing and requires no intermediate message storage while the routing decision is being made, the traditional concept of maximum path length becomes a moot issue. To a great extent, path length determines network response time for a message. With the spiral topology, if simulation analysis yields a response time that is too high due to long paths or queues (or whatever the reason), one simply increases link speed to the point where the queues become shorter, and the response time is again bearable. Once this fine tuning using a simulation or analytical model yields an acceptable response time, network analysts can then implement that configuration.

5.4. Simulation Model Summary

The model used to simulate the spiral computer network topology contains approximately 2700 lines of C program source code (Appendix E). Spiral networks containing 4 to 20 modules (16 to 80 nodes) were established and evaluated, using 0, 1, 2, and 3 failed nodes. Also, the failure of complete modules was analyzed. In every case, the common link speed was 19200 bits per second; 4000 messages were delivered to reach a simulated steady state condition; and 4000 messages were then delivered and used to gather statistics. Appendix B contains more details on the design and implementation of this simulation model. And Appendix D contains the Summary of Simulation Results for each of the simulation runs.

The computer model used to gather statistics for comparison simulates the spiral network topology as described in the current chapter. The performance of the topology was evaluated using established techniques that apply to networks whose messages follow a Poisson arrival pattern, with exponentially distributed lengths. Analytical results including mean queue length, mean and maximum path length, mean network response time, and system utilization, were compared to the values obtained through simulation.

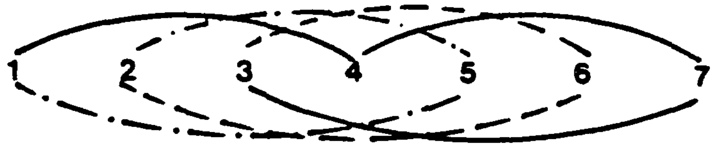
6. ROUTING IN A SPIRAL NETWORK

6.1. Introduction

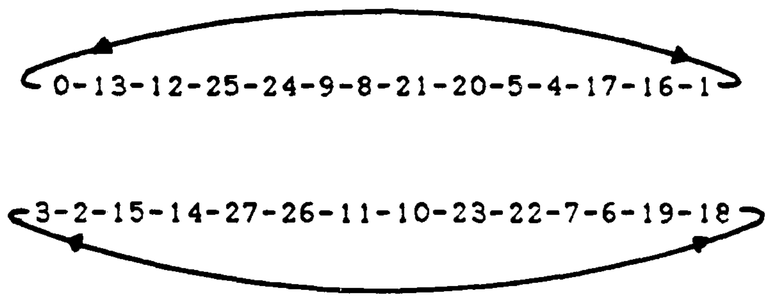
The next sections discuss how the spiral routing algorithm transports messages from source to destination nodes. These sections include examples of routing with and without failures. The failure scenarios range from a single failure, to the case where some nodes are not reachable. All examples use the seven module, 28 node spiral topology for convenience. Also, the same source-destination pair is assumed.

6.2. The Routing Algorithm

The spiral network connectivity pattern generates two separate top and bottom spirals along which messages travel to their destination. Figures 6-1(a) and (b) duplicate the network threading and unwound spiral patterns for the seven module, 28 node network. While this discussion assumes a seven module, 28 node network, it applies to any size spiral network. Messages whose destination address ends in "0" or "1", base 4, are routed along the top spiral. Those ending in "2" or "3" use the bottom spiral. These four possibilities led to our selection and use of the base 4 numbering system to label nodes. The Preprocessing Algorithm at the source node determines the optimum path to



a). Seven Module Network Threading Pattern.



b). Unwound Spirals.

Figure 6-1. Threading and Unwound Spiral Patterns.

destination by finding the spiral path which contains the fewest intermediated nodes, whether to the right or left. A one bit spiral flag (SF) and one bit direction flag (DF) are then initialized by the Preprocessing Algorithm before the message enters the network. SFDF pairs have the following meaning: 00 sends the message along the top spiral to the right; 01 uses top spiral left; 10 sends it bottom spiral right; and 11 uses the bottom spiral left. These flags are attached to the message header, and remain unchanged as long as no faulty links or nodes are encountered. When a message arrives at an intermediate node, the Rapid Processing Algorithm places it on the proper output queue from that node, based on the SFDF pair value. If the selected output queue in the message's path is connected to a failed link or node, then the Rapid Processing Algorithm complements the SF. If the SF has already been complemented within the current module (as indicated by a module spiral change counter), then a spiral change has already been attempted. So the DF is complemented. This case suggests that failures in the current direction have occurred along both the top and bottom spirals at or near the same point, such that these spirals are completely severed in the current direction. For if this failure pattern did not exist, then the message would advance at least one node closer to its destination. Making such an advance would move the message past the point of failure on the other spiral, and allow it to continue on towards the final destination.

After the DF is changed, the message then continues on towards its destination in the opposite direction. Thus, based on the pattern of failed nodes, it is possible for a message to retrace its steps back through the source node as it seeks out the path in the opposite direction. The routing procedure must check to ensure messages do not oscillate between spirals, or alternate directions indefinitely. Fault tolerance is inherent in the algorithm since altering the SFDF pair is guaranteed to find a path to destination if one exists, regardless of the pattern of failure(s).

The routing algorithm used in our spiral topology is summarized as follows:

1. If the message is a new arrival to the network, the spiral and direction flags are initialized by the Preprocessing Algorithm at the source node.

2. The message arrives at a node and is checked to determine if it has reached its destination module. If so, it is gated by the Rapid Processing Algorithm to the destination node within the current module.

3. If this is not the destination module, the message is sent to its gateway node within the current module if that node is operating. The last digit of this gateway node's address is SFDF, base 4. Go to step #7.

4. If the gateway node has failed and the SF has not been changed while in the current module, then SF is complemented. Return to step #3.

5. If the gateway node has failed, the SF has already been

changed within the current module, and the DF has not, then complement the DF. Return to step #3.

6. If the gateway node has failed, the SF has already been changed within the current module, the DF has already been changed, and the message has returned to the original source module twice, then there is no path to the destination. End of trip for this message. (The algorithm terminates for this message. Take appropriate action to notify source node of inability to deliver.)

7. Once at the gateway node, gate the message to the module in its route path. Return to step #2.

6.3. Examples of How the Routing Algorithm Operates

All examples used in this section were chosen to emphasize spiral's high tolerance to failure(s). While the response time is increased as more failures occur (to a point, then the response time again decreases), our primary concern is confirmation that the spiral topology still functions in the case of failures. Response time is addressed in more detail in the next chapter.

Consider the seven module, 28 node network repeated in figure 6-2. Assume no failures, and that we are sending a message from source node 20 (110 base 4), to destination node 27 (123 base 4). The Preprocessing Algorithm determines the optimum path and sets the SFDF pair to 11 (bottom spiral left). The route chosen by the algorithm will be 110 -> 113 -> 22 -> 23 -> 122 -> 123.

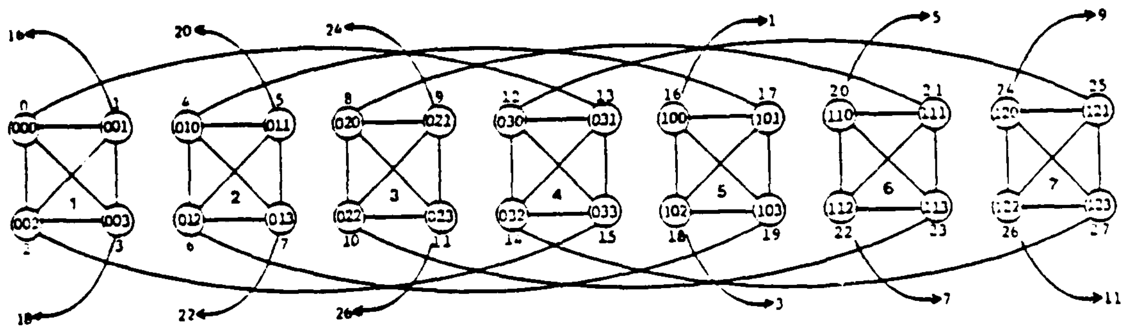
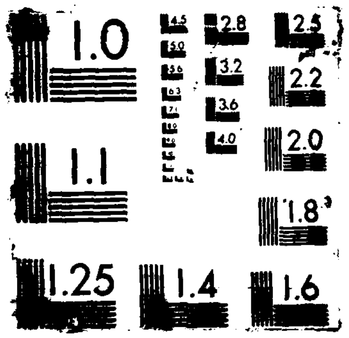


Figure 6-2. Seven Module Spiral Network, No Failures.

AD-A196 115   A FAULT TOLERANT SELF-ROUTING COMPUTER NETWORK TOPOLOGY   2/3
(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH
T L MITCHELL 1988 AFIT/CI/NR-88-68

UNCLASSIFIED   F/G 12/7   NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

Next, the same source-destination pair is assumed, but now node 11 (23, base 4) has failed. Refer to figure 6-3 for this example. The route now chosen by the algorithm is 110 -> 113 -> 22 -> 21 -> 120 -> 123.

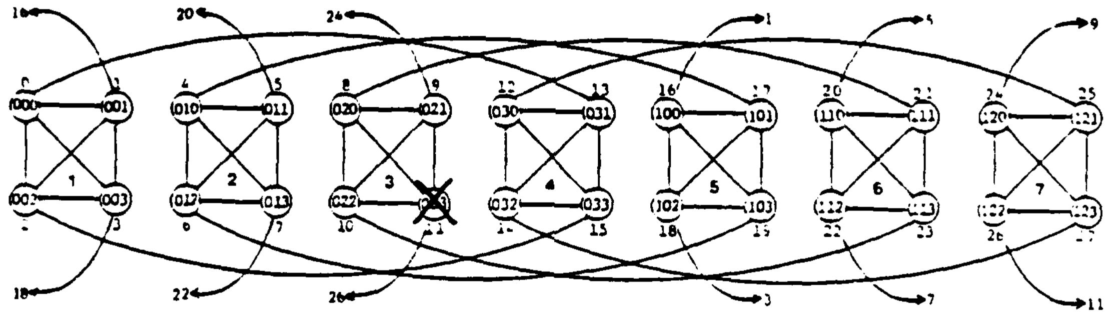


Figure 6-3. Single Arbitrary Node Failure.

In this next example, nodes 9 and 11 have failed (nodes 21 and 23, base 4). We again assume the same S-D pair of 20 - 27. From figure 6-4, we see that the path chosen by the Routing Algorithm is 110 -> 113 -> 22 -> 113 -> 112 -> 13 -> 12 -> 103 -> 102 -> 03 -> 02 -> 33 -> 32 -> 123. Notice in this case, both spirals are cut off in the direction of the shortest path. The algorithm adjusts on-the-fly by complementing both SF and DF. The result is the message retracing its steps back through the source module in the opposite direction. However, the destination was still reachable.

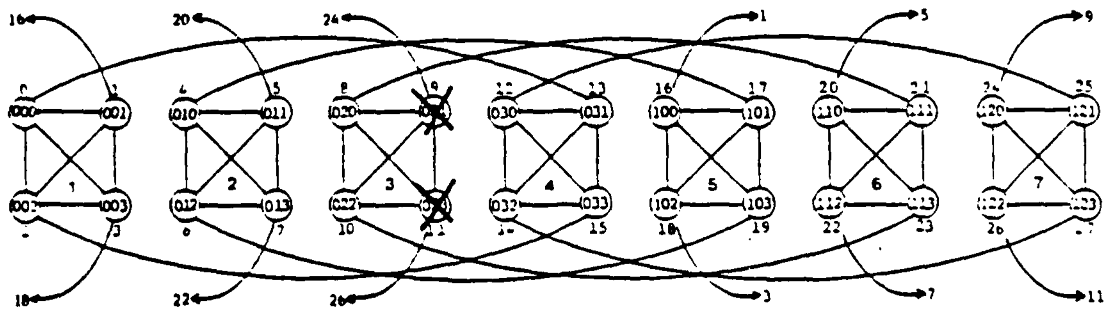


Figure 6-4. Two Node Spiral Network Failure.

Next, we start with the last example where both spirals were cut off in the same direction, and additionally kill node 03. Again, we use the source-destination pair 20 - 27. The path chosen can be traced through figure 6-5 as 110 -> 113 -> 22 -> 113 -> 112 -> 13 -> 12 -> 103 -> 102 -> 100 -> 01 -> 00 -> 31 -> 30 -> 121 -> 123. Again, note that the destination is still reachable.

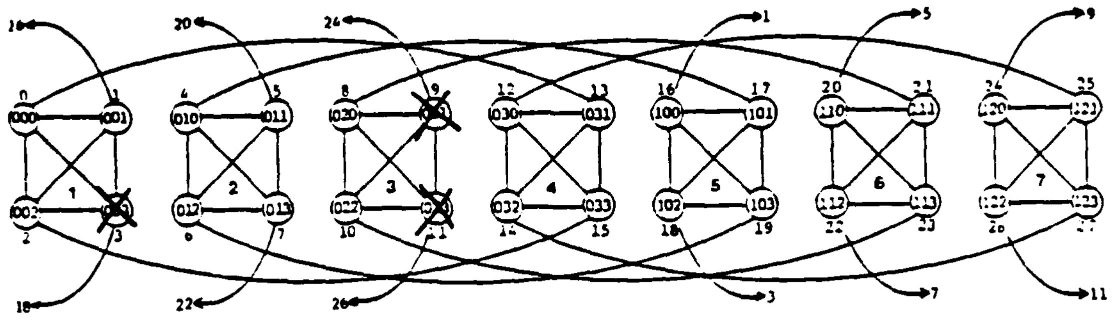


Figure 6-5. Three Node Spiral Network Failure.

In this final example, the failure pattern is such that the destination is unreachable. In addition to all previous failures, we also intentionally kill node 01 (refer to figure 6-6 for the pictorial representation). In this last case, the self-routing algorithm results in the following path: 110 -> 113 -> 22 -> 113 -> 112 -> 13 -> 12 -> 103 ->

102 -> 100 -> 101 -> 10 -> 11 -> 110. But now, both spirals have been attempted in both directions, and the message has returned to the source module (#6) twice. The algorithm recognized that the destination is indeed unreachable, and therefore terminated at step #6. Why should the algorithm allow the message to return to the source module twice? Because one return trip is possible due to direction change, as occurred in the scenario preceding this current example. If node 00 instead of 01 had failed (figure 6-7), then the destination would still be reachable, since a spiral change would again occur. In that case, the path would be 110 -> 113 -> 22 -> 113 -> 112 -> 13 -> 12 -> 103 -> 102 -> 100 -> 01 -> 02 -> 33 -> 32 -> 123.

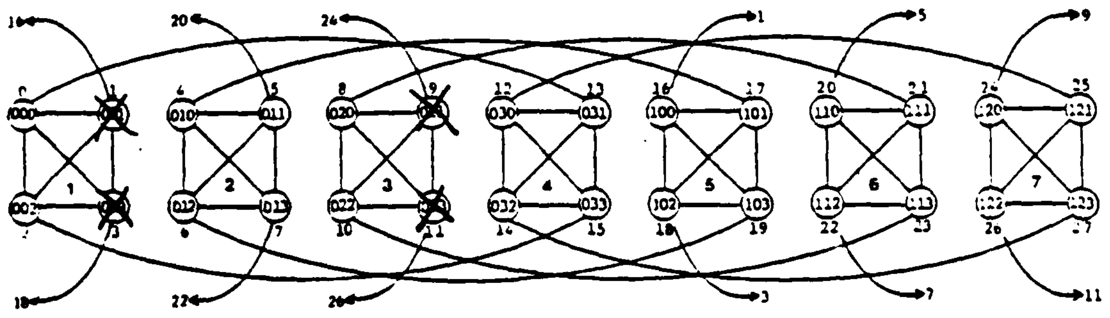


Figure 6-6. Unreachable Destination Example.

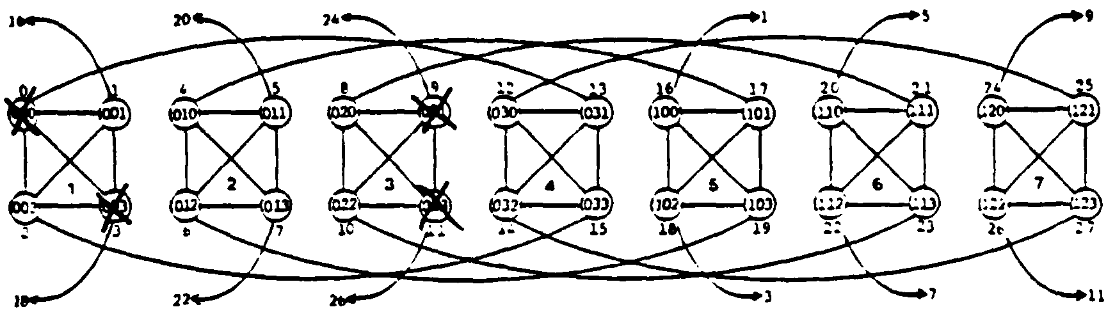


Figure 6-7. Destination Reachable After Several Changes.

6.4. Summary

In each of the above examples, the resulting path chosen by the algorithm was longer than the optimum one of the error free case. As mentioned earlier, the longer path means increased delay for the message. However, the achievement of high tolerance to failures anywhere in the network is our primary concern. Further, link speeds are arbitrarily selected to achieve specified levels of performance. Network planners need only select a higher link speed for the error free network to compensate for the increased path length resulting from severe node or module failures. Finally, for any source-destination pair under error free conditions, the algorithm always selects the optimum path. In the presence of errors, if a path exists between any two nodes, the algorithm is guaranteed to find it.

7. ANALYSIS OF THE ERROR FREE SPIRAL NETWORK TOPOLOGY

7.1. Introduction

This chapter's primary emphasis is on reporting results that apply to the error free spiral network topology. When failed nodes exist in a spiral network, closed form algebraic results similar to the ones reported in this chapter, are either impossible, or extremely difficult to derive. For example, the impact of failures on the mean path length is dependent on the location of those failures. Two arbitrary node failures on opposite spirals that are not directly opposite each other, have less impact on the mean path length than two failures that are opposite each other. In the latter case, both top and bottom spirals are cut off, causing a message to reverse its direction and retrace the path traversed to that point. This reversal by messages that need to advance beyond the point of the two failures will result in a longer mean path. Even if a closed form expression for the mean path length could be found that compensates for each possible failure pattern, the resulting expression would be extremely complicated, and add little to the overall understanding of the network's performance. The combinations of failure patterns are numerous, because the number of failures can range from 1 to as many nodes as there are in the network. Even if only two or three remaining good nodes can reach each other, theoretically, there still is a mean path length.

Therefore, the approach to analyzing the spiral topology when nodes have failed is to compare the results for no failures to the same size spiral network containing 1, 2, or 3 failures. We also analyze the impact of the failure of a single, or several modules, on the overall network. A thorough analysis of the spiral network topology when arbitrary nodes and complete modules have failed, is found in the next chapter.

Theoretical results in this current chapter are compared to those obtained from simulation. The simulation model used to generate these comparative results contains approximately 2700 lines of C program source code (Appendix E). Spiral networks containing 4 to 20 modules (16 to 80 nodes) were established and evaluated. In every case, the common link speed was 19200 bits per second; 4000 messages were delivered to reach a simulated steady state condition; and 4000 messages were used to gather statistics. Appendix B contains more details on the design and implementation of this simulation model. And Appendix D contains the Summary of Simulation Results for each run.

7.2. Theorem on Disjoint Paths

THEOREM 1: In any size spiral network with no failed nodes or links, there exists four disjoint paths between any source-destination pair.

Proof: In the spiral connectivity scheme, every third module is threaded together top and bottom in a circular fashion. Nodes on the front end of each module are connected to the nodes on the rear side of their link module. This pattern results in two spirals of equal length: one on top and the other on bottom. Since these spirals are circular, the four disjoint paths that result are 1) top spiral traveling to the right; 2) top spiral left; 3) bottom spiral right; and 4) bottom spiral left.

Q.E.D.

As an example, consider the seven module, 28-node network (figure 5-4). Using the source-destination pair 5 -> 18, the four disjoint paths using base 10 numbers are:

5 > 20 > 21 > 8 > 9 > 24 > 25 > 12 > 13 > 0 > 1 > 16 > 18

5 > 7 > 22 > 23 > 10 > 11 > 26 > 27 > 14 > 15 > 2 > 3 > 18

5 > 4 > 17 > 18

5 > 6 > 19 > 18.

7.3. Theorem on Maximum Path Length

THEOREM 2: The maximum path length in any size spiral network having no failed nodes or links, measured in hops under the shortest path algorithm, is equivalent to the number of modules contained in that network.

Proof: Define the following variables:

m	= number of modules in the network,
n	= number of nodes in the network,
s	= number of nodes on each of the two spirals, and
h	= maximum number of hops between the most distant source-destination pair.

Since the minimum four module, 16 node spiral network is expanded by modules of four nodes each, $n = 4 * m$. Further, since there are two equal disjoint spirals in a network, $s = n/2$. Consider one spiral. Since $4 * m$ is always even, $s = (4 * m)/2$ is also even. The maximum number of hops, h, between any two most distant of 2m nodes is $h = (2m)/2 = m$. For if h were greater than $(2m)/2$, then choosing to send the message in the opposite direction along this spiral results in a path shorter than $h = m$. Even if the source-destination nodes are on opposite spirals, direct connection to the appropriate link node on the other spiral does.not lengthen the maximum path. Thus

$$h = \frac{s}{2} = \frac{n/2}{2} = \frac{n}{4} = \frac{4 * m}{4} = m.$$

Q.E.D.

7.3.1. Analysis of Maximum Path Length

The following analysis compares theorem 2's formula results to those of the simulation model. In the model, if the path of the most recently delivered message is longer than any previous path, then this length becomes the updated maximum path value.

Case 1: $m = 4$

theorem: $h = m = 4.$

simulation: $h = 4.$

Case 2: $m = 5$

theorem: $h = m = 5.$

simulation: $h = 5.$

Case 3: $m = 7$

theorem: $h = m = 7.$

simulation: $h = 7.$

Case 4: $m = 8$

theorem: $h = m = 8.$

simulation: $h = 8.$

Case 5: m = 10

theorem: h = m = 10.

simulation: h = 10.

Case 6: m = 11

theorem: h = m = 11.

simulation: h = 11.

Case 7: m = 13

theorem: h = m = 13.

simulation: h = 13.

Case 8: m = 14

theorem: h = m = 14.

simulation: h = 14.

Summary: When comparing the theoretical result of maximum path length to that obtained from the simulation model, the results are always identical. This is true no matter the size of the spiral network under study.

7.4. Theorem on Traffic Between any Source-Destination Pair

Theorem 3: Let γ_{ij} messages per second represent the average amount of external traffic entering node (i) and destined for node (j). Assuming that each node sends this new traffic to all other nodes with equal probability, then for any size spiral network with or without failed nodes,

$$\gamma_{ij} = \frac{4(n - f)}{n(n-1)IAT} .$$

Where:

n = the number of nodes in the network,

IAT = mean time (in seconds) between arrival of external messages to each of the four links at node (i), and

f = number of failed nodes in the spiral network.

Proof (this result is used in sections 7.6 and 7.10): External messages arrive at each link with frequency 1/IAT. Since there are four links attached to each node, the arriv l rate of external messages to each node is 4/IAT. Further, since each node sends to all other nodes with equal probability, the proportion of this new traffic sent to each node is 1/(n-1). When there are no failures, the average amount of newly generated traffic sent from source node (i) to destination node (j) per second is:

$$\gamma_{ij} = \frac{4}{IAT} * \frac{1}{n-1} = \frac{4}{(IAT)(n-1)}.$$

In the spiral topology, there is no global node status information, thus good nodes will send messages to failed ones, not knowing that the destination has failed. However, failed nodes send no messages. Therefore (n-f)/n represents the proportional impact that failures have on externally generated traffic. The reduction in external traffic as a result of failures is 1 - (n - f)/n.

Incorporating the proportional impact of failures into the above equation, we now get

$$\gamma_{ij} = \frac{4}{(n-1)(IAT)} * \frac{n-f}{n} = \frac{4(n-f)}{n(n-1)(IAT)}.$$

Q.E.D.

7.5. Theorem on Expected Average Link Traffic

7.5.1. Introduction

The previous theorem is concerned with the arrival of _external_ traffic to the _four_ links that comprise a node, and how that new traffic is distributed to each of the remaining (n - 1) nodes. The next theorem addresses a single (_one_ of the four) arbitrary link at a node. The theorem result is an expression for determining the expected average amount of traffic that is sent over any arbitrary link. That traffic includes both newly generated external traffic and transient messages.

Theorem 4: Let λ_i messages per second represent the expected average amount of newly generated and transient traffic passed over an arbitrary link (i). Then in any size spiral network which has no failed nodes or links,

$$\lambda_i = \frac{1}{IAT} + .375\beta,$$

where:

IAT = mean time (in seconds) between arrival of external messages to each link, and

β = the average amount of transient traffic entering any node.

7.5.2. Preliminary Discussion

If we start with some source node (i), then the traffic from node (i) to directly connected node (j) consists of new external traffic that enters the network at node (i) destined for (j), plus transient traffic that uses the link between nodes (i) and (j) as an intermediate hop. When this combined traffic enters node (j), it is joined by new traffic that enters the network at node (j), plus a portion of the transient traffic that enters node (j) on the other links tied into node (j). A portion of this new total amount of traffic then passes over the next link to node (k), where the process of combining new external and transient traffic is repeated. This analysis suggests that the average amount of traffic on an arbitrary path increases proportional to the path length. But since every node in a spiral network falls near the end of the path from some source node, we would expect this proportional increase in traffic to be distributed uniformly throughout the network. In the spiral topology, the path taken by an arbitrary message is part of a tandem network that contains cross links at each node. So while transient messages at each node joins the flow of newly generated traffic, at some point along the path, we assume that an amount of transient traffic equivalent to that which joined the path, will again leave, and become part of some other path in the network. Therefore, the reasonable assumption is made that the average transient traffic (β) arriving at an arbitrary link to join newly generated messages is a constant. This assumption and discussion of the behavior of queues in tandem that merge with cross links parallels the analysis of tandem queues in [42], where the assumption is made that transient messages move into a path at some node, and then leave that path at the next node.

There are four unique cases that must be analyzed when considering the mean traffic over a link. These cases exist because we have four links connected to each node. The analysis that follows is based on a fully connected module, using variables defined as follows:

λ_i = expected mean traffic (in seconds) passed over any arbitrary link (i),

IAT = mean time (in seconds) between arrival of external messages to each of the four links at a node, and

β = the average amount of transient traffic per second entering any arbitrary node.

In each case analyzed, the solid lines with direction arrows in figures 7-1 through 7-4, highlight the links that may impact the traffic that helps form λ_i. The broken lines are included to complete the module connectivity. Also, in each of the four cases that follows, three links (with direction arrows in the figures) lead into the node, and one link leads away from it. We arbitrarily choose node (1) as the source node. The values of λ_2, λ_3, and λ_4 in figures 7-1 through 7-4 represent the traffic (in seconds) sent from nodes (2), (3), and (4) respectively.

Case 1: Node (1) sends to node (4).

In this first case (figure 7-1), a portion of the transient traffic β from the external link is joined by the new traffic generated at the link feeding node (4) from node (1). So the value of λ_i is determined by the rate of newly arriving messages to the node (1) to node (4) link, plus a portion of the through traffic β. None of λ_2 or λ_3 is fed to λ_i, since traffic sent to node (4) from nodes (2) and (3) uses the directly connected node (4) links. The amount of β

going to λ_i depends on how much terminates at node (1), and how much is sent to nodes (2) and (3).

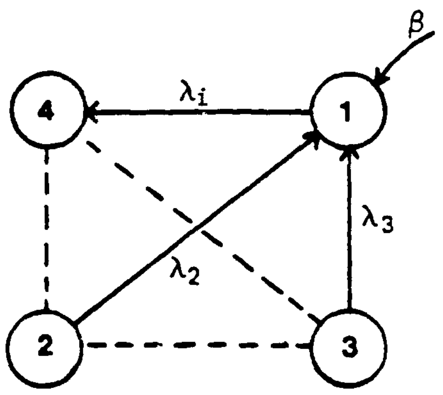


Figure 7-1. Node (1) Sends to Node (4).

Case 2: Node (1) Sends to Node (2).

In our second case, the value of λ_i is determined by the rate of newly arriving messages to the link feeding node (2) from node (1), plus the portion of the through traffic from

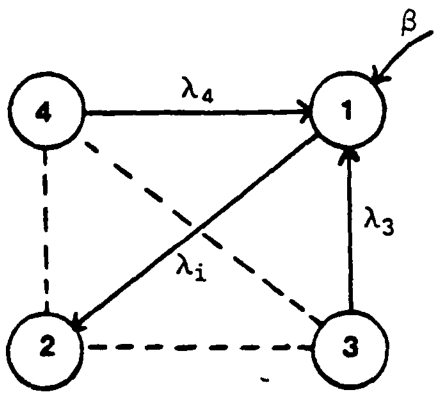


Figure 7-2. Node (1) Sends to Node (2).

β (see figure 7-2). Again, none of λ_3 or λ_4 is fed to λ_i, since traffic sent to node (2) from nodes (3) and (4) uses the directly connected node (2) links. The amount of β going to λ_i depends on how much terminates at node (1), and how much is sent to nodes (3) and (4).

Case 3: Node (1) Sends to Node (3).

Now, the value of λ_i is determined by the rate of newly arriving messages to the link feeding node (3) from node (1), plus the portion of the through traffic from β (see figure 7-3). As before, none of λ_2 or λ_4 is fed to λ_i, since traffic sent to node (3) from nodes (2) and (4), uses the directly connected node (3) links. The amount of β going to λ_i depends on how much terminates at node (1), and how much is sent to nodes (2) and (4).

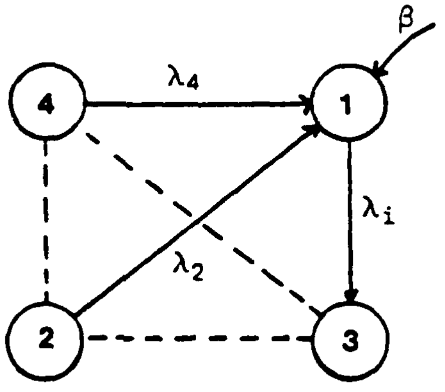


Figure 7-3. Node (1) Sends to Node (3).

Case 4: Node (1) Sends to Distant Node.

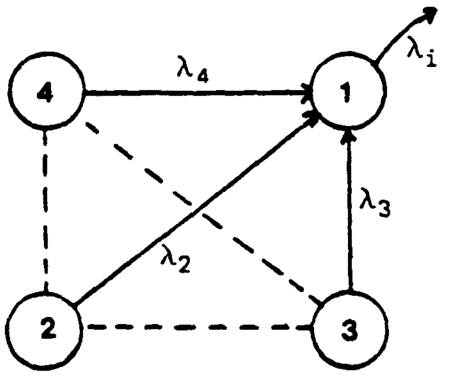


Figure 7-4. Node (1) Sends to Distant Node.

In this final case, node (1) sends to a node that is a part of the directly connected next (or previous) module. The value of λ_i is now determined by the rate of newly arriving messages to the link feeding the distant node from node (1), plus the portion of traffic from λ_2, λ_3 and λ_4 ($\lambda_2 = \lambda_3 = \lambda_4 = \beta$). The amount of λ_2, λ_3, and λ_4 that helps to form λ_i is determined by whether or not node (1) is the final destination node for a message sent from or through nodes (2), (3), or (4).

We now form mathematical representations of these four cases, and combine them to conclude the proof of theorem 4.

7.5.2. Proof of Theorem 4

Case 1: In this first case, β either terminates at node (1), or is sent on to nodes (2), (3), or (4). β therefore goes to node (4), and thus helps form λ_i, with probability .25:

$$\lambda_i = \text{arrival rate of newly generated messages} + \text{portion of through traffic}$$

$$\lambda_i = \frac{1}{IAT} + [\text{Probability}(\beta \text{ goes to } \lambda_i)]\beta$$

$$\lambda_i = \frac{1}{IAT} + .25\beta. \tag{7.5.1}$$

Case 2: Again, β either terminates at node (1), or is sent on to nodes (2), (3), or (4). The probability of β helping to form λ_i is .25 as before.

$$\lambda_i = \text{arrival rate of newly generated messages} + \text{portion of through traffic}$$

$$\lambda_i = \frac{1}{IAT} + [\text{Probability}(\beta \text{ goes to } \lambda_i)]\beta$$

$$\lambda_i = \frac{1}{IAT} + .25\beta. \tag{7.5.2}$$

Case 3: This case is the same as the two previous ones: β either terminates at node (1), or feeds nodes (2), (3), or (4) with equal probability.

$$\lambda_i = \text{arrival rate of newly generated messages} + \text{portion of through traffic}$$

$$\lambda_i = \frac{1}{IAT} + [\text{Probability}(\beta \text{ goes to } \lambda_i)]\beta$$

(7.5.3) $$\lambda_i = \frac{1}{IAT} + .25\beta.$$

Case 4: In this last case, λ_2 terminates at node (1) or feeds λ_i, λ_3 terminates at node (1) or feeds λ_i, and λ_4 terminates at node (1) or feeds λ_i. Each of these terminations occur with probability .5.

λ_i = arrival rate of newly generated messages
+ portion of through traffic

$$\lambda_i = \frac{1}{IAT} + [\text{Probability}(\lambda_2 \text{ or } \lambda_3 \text{ or } \lambda_4 \text{ goes to } \lambda_i)]\beta$$

$$\lambda_i = \frac{1}{IAT} + [P(\lambda_2) + P(\lambda_3) + P(\lambda_4) - P(\lambda_2 \cap \lambda_3) - P(\lambda_2 \cap \lambda_4) - P(\lambda_3 \cap \lambda_4)]\beta$$

$$\lambda_i = \frac{1}{IAT} + [.5+.5+.5-(.5)(.5)-(.5)(.5)-(.5)(.5)]\beta$$

$$\lambda_i = \frac{1}{IAT} + [.5(3) - (.5)(.5)(3)]\beta$$

(7.5.4) $$\lambda_i = \frac{1}{IAT} + .75\beta.$$

Consolidating equations 7.5.1 through 7.5.4 for the four cases and simplifying, yields the result we seek:

$$\lambda_i = (\frac{1}{IAT} + .25\beta)(.75) + (\frac{1}{IAT} + .75\beta)(.25)$$

$$\lambda_i = \frac{.75}{IAT} + .1875\beta + \frac{.25}{IAT} + .1875\beta$$

$$\lambda_i = \frac{1}{IAT} + .375\beta.$$

Q.E.D.

7.5.3. Analysis of Expected Average Traffic

Table 7-1 contains analytical results obtained from applying the formula in theorem 4 to various size spiral networks for selected values of β. For each spiral network analyzed, the overall system utilization was first found by ignoring transient traffic ($\beta = 0$). Then we assumed that transient traffic joined newly arriving traffic at a rate of .05 messages per second ($\beta = .05$). The value for the arrival rate of new messages (1/IAT) was also .05, since IAT = 20 seconds. The column in table 7-1 which reports theoretical system utilization (ρ_t) contains results obtained from applying the formula to find λ_i based on β and IAT.

Assuming the size of a newly arriving message is chosen from an infinite population that is distributed exponentially with mean $1/\mu$ bits per message, and that arrivals follow a Poisson arrival rate with mean λ_i, a very good approximation for ρ_t at this point is the mean utilization for a single link (based on λ_i calculated using theorem 4), multiplied by the mean path length (L_m) as found in the simulation model (i.e. $\rho_t = \lambda_i L_m / \mu C$) [42]. Theorem 5 and analytical work in section 7.8 lead to the exact expression for ρ_t. Section 7.8 also confirms the accuracy of the current approximation. The last column in table 7-1 contains the measured utilization found in the model, calculated as follows:

$$\rho_m = \frac{\text{average time all links are busy}}{\text{total network time}}.$$

Notice that without exception, when transient traffic is ignored ($\beta = 0$), the theoretical utilization (ρ_t) is closest to the measured result (ρ_m). Even a small amount of transient traffic (β = .05) causes theoretical utilization to surpass the measured values. In concluding this analysis based on the above theorem, without exception, the analysis of theoretical versus simulation results confirm that transient traffic can indeed be ignored when calculating the system utilization under the assumed conditions. The fact that when $\beta = 0$, theoretical results are closest to simulation values, supports the long accepted assumption that for analyzing M/M/1 queues connected in tandem, transient traffic can be ignored [42]. Ignoring transient traffic is possible because the rate at which transient traffic flows into that tandem network equals the rate at which it flows out of it. So while the formula in theorem 4 is a precise representation of the expected average traffic arriving at an arbitrary link, a more simple procedure that is just as accurate is to use the rate at which new messages arrive at the links, ignoring the transient traffic.

Table 7.1. Average Traffic Result Comparisons.

modules	β	λ_i	ρ_t	ρ_m
4	.00	.05000	.04984	.04820
	.05	.06875	.06853	
5	.00	.05000	.06043	.05890
	.05	.06875	.08309	
7	.00	.05000	.08078	.07972
	.05	.06875	.11107	
8	.00	.05000	.09060	.08849
	.05	.06875	.12458	
10	.00	.05000	.11166	.11286
	.05	.06875	.15353	
11	.00	.05000	.12308	.12104
	.05	.06875	.16923	
13	.00	.05000	.14425	.14673
	.05	.06875	.19834	
14	.00	.05000	.15312	.15060
	.05	.06875	.21054	
16	.00	.05000	.17561	.17136
	.05	.06875	.24147	
17	.00	.05000	.18580	.17796
	.05	.06875	.25548	
19	.00	.05000	.20473	.20226
	.05	.06875	.28150	
20	.00	.05000	.21251	.21263
	.05	.06875	.29220	

7.6. Theorem on Total One Way Link Traffic

7.6.1. Introduction

In the previous section, we presented as theorem 4 an expression for calculating λ_i, the expected average amount of external and transient traffic (in messages per second) passed over any arbitrary link (i). If we form the sum of all the one way λ_i's in a network, then we arrive at the total one way link traffic in that network. The next theorem provides an alternate exact expression for finding the total average one way link traffic. Theorem 5 will be applied in sections 7.7 and 7.8.

Theorem 5: The total average one way link traffic (λ) for any size spiral network having no failed nodes or links, is

$$\lambda = \frac{4}{IAT(n-1)} \left\{ \frac{n}{2} \left[5m-4 + \sum_{i=1}^{\lceil \frac{m}{2} \rceil - 1} (7m-14i-3) + \sum_{i=1}^{\lfloor \frac{m}{2} \rfloor - 1} (m-2i-1) \right] + mn \right\},$$

where:

IAT = mean time in seconds between arrival of external messages to each link,

n = the total number of nodes in the network,

m = the total number of modules in the network ($m = n/4$).

7.6.2. Preliminary Discussion

We now know from theorem 4 that transient traffic entering a network of tandem queues can be ignored, since the flow rate into that tandem link equals the flow rate out. Even if the transient traffic was not ignored, on the average, it would be constant, and thus could be removed from computations. The theorem just presented as theorem 5 pertains to _external_ traffic only.

There are three types of links in any size spiral network:

Type I: Gateway Links - These links connect the modules together to help form the top and bottom spirals. Examination of any size spiral network topology quickly confirms that there are (n) Type I links in each full-duplex spiral network. So the one way total of Type I links is n/2.

Type II: Transient Links - These links form bridges for the gateway links, and complete the connections that form the top and bottom spirals. Type II links are the top and bottom connections in each module. Again, examination of any size spiral network topology confirms that there are also n/2 Type II links in each one way spiral network.

Type III: Crossover Links - These links, along with the Type II Transient Links, form the individual modules. The crossover links cause a change from one spiral to the other. The number of crossover links in any full duplex spiral network is quickly verifiable to equal twice the number of nodes present in that network. Thus, there are (n) Type III links in each one way spiral network.

The total number of one way links in any size spiral network, then, is the sum of these three types.

The external traffic load passed over a particular link depends on the mean message arrival rate to the link, and whether the link is of Type I, II, or III. The heaviest links are of Type I and II since they form the top and bottom spirals. In an error free network, only links directly connected to source or destination nodes perform as Type III. In other words, a message uses a Type III link only at the beginning or end of its path. As it moves along the spiral, the message spends the remainder of the transmission time on Type I or II links. Let λ_I, λ_{II}, and λ_{III} represent the total number of combinations of each type of link in any size spiral network. Recall from theorem 3 in section 7.4 that γ_{ij} in messages per second, represents the amount of new external traffic entering node (i) and destined for node (j). Then the amount of one way link traffic on each of the three types of links is defined as follows:

TYPE I: $\gamma_{ij}\lambda_{I} = \lambda_1$.

TYPE II: $\gamma_{ij}\lambda_{II} = \lambda_2$.

TYPE III: $\gamma_{ij}\lambda_{III} = \lambda_3$.

7.6.3. Proof of Theorem 5

For a Type I link, if one listed all the possible one way traffic combinations (γ_{ij}'s) for each source-destination pair, the total number results in the following pattern:

$$\frac{m}{2} + (m-0) + 3(m-1) + (m-2) + 3(m-3) + (m-4) + 3(m-5)\ldots$$

$$= \frac{m}{2} + \sum_{i=0}^{\lceil \frac{m}{2} \rceil - 1} (m-2i) + \sum_{i=0}^{\lceil \frac{m}{2} \rceil - 1} 3(m-2i-1), \text{ or}$$

$$\lambda_I = \frac{m}{2} + \sum_{i=0}^{\lceil \frac{m}{2} \rceil - 1} (4m-8i-3).$$

Where:

m = the number of modules in the network, and

$\lceil\rceil$ = the ceiling function. $\lceil m/2 \rceil = m/2$ if m is even, else round up to the next integer.

Listing all the possible one way traffic combinations (γ_{ij}'s) for each Type II source-destination pair results in the next pattern:

$$\frac{m}{2} + (m-0) + (m-1) + 3(m-2) + (m-3) + 3(m-4) + (m-5)\ldots,$$

$$\lambda_{II} = \frac{m}{2} + \sum_{i=1}^{\lceil \frac{m}{2} \rceil - 1} 3(m-2i) + \sum_{i=0}^{\lfloor \frac{m}{2} \rfloor - 1} (m-2i-1).$$

Where $\lfloor\,\rfloor$ is the floor function. $\lfloor m/2 \rfloor = m/2$ if m is even, else truncate the decimal portion.

And finally, it is quickly verifiable that the number of one way γ_{ij}'s of Type III is exactly equal to the number of modules (m) in the network. From theorem 3, with no failures (f = 0), the average amount of external traffic from source node (i) to destination (j) is $\gamma_{ij} = 4/(IAT(n-1))$. Therefore the total average one way traffic is:

$$\lambda = \gamma_{ij} \sum_{i=1}^{n/2} \lambda_I + \gamma_{ij} \sum_{i=1}^{n/2} \lambda_{II} + \gamma_{ij} \sum_{i=1}^{n} \lambda_{III}$$

$$= \gamma_{ij}\left[\sum_{i=1}^{n/2} \lambda_I + \sum_{i=1}^{n/2} \lambda_{II} + \sum_{i=1}^{n} \lambda_{III} \right]$$

$$= \gamma_{ij}\left[\frac{n}{2}(\lambda_I + \lambda_{II}) + n\,\lambda_{III} \right]$$

$$\lambda = \frac{4}{IAT(n-1)} \left\{ \frac{n}{2}\left[5m-4 + \sum_{i=1}^{\lceil \frac{m}{2} \rceil - 1} (7m-14i-3) + \sum_{i=1}^{\lfloor \frac{m}{2} \rfloor - 1} (m-2i-1) \right] + mn \right\}.$$

Q.E.D.

7.7. Theorem on Mean Path Length

THEOREM 6: The mean path length (L) in any size spiral network having no failed nodes or links, and measured in hops, is given by

$$L = \frac{h}{n-1}(2h + 1),$$

where:

h = maximum number of hops between the most distant source-destination pair, and

n = number of total nodes in the network.

Proof:

From theorem 1, there are four disjoint paths between any source-destination pair. Starting at an arbitrary source node, there are four possible links for the first hop. From any intermediate node (i), there are also four options to form the (i + 1)st link. So for the first (h - 1) links, the mean number of hops is

$$\sum_{i=1}^{h-1} \frac{4}{n-1} i. \tag{7.7.1}$$

Now for the last link, there are only three remaining links, since two directly connected links are tied to the same destination node. Mathematically, this last hop is represented as

$$\frac{3}{n-1} h. \tag{7.7.2}$$

Combining the results from equations (7.7.1) and (7.7.2), we form the weighted mean path length, where the weight is the number of the link in a path.

$$L = \sum_{i=1}^{h-1} \frac{4}{n-1}\, i + \frac{3}{n-1}\, h$$

$$L = \frac{1}{n-1} \left[\sum_{i=1}^{h-1} 4i + 3h \right]$$

$$L = \frac{1}{n-1} \left[4 \sum_{i=1}^{h-1} i + 3h \right].$$

Now, using the well known fact that $\sum_{i=1}^{n} i = \frac{n}{2}(n + 1)$,

$$L = \frac{1}{n-1} \left[4 \frac{(h - 1)(h)}{2} + 3h \right]$$

$$= \frac{1}{n-1} \left[2(h^2 - h) + 3h \right]$$

$$= \frac{1}{n-1} \left[2h^2 - 2h + 3h \right]$$

$$= \frac{h}{n-1} (2h + 1)$$

Q.E.D.

As an example of how the theorem is applied, consider the 7-module, 28-node spiral network. Arbitrarily choose node (0) as the source node. The four disjoint paths are:

<u>Source</u>							<u>Farthest node from source</u>
0 ->	1 ->	16 ->	17 ->	4 ->	5 ->	20 ->	21
0 ->	3 ->	18 ->	19 ->	6 ->	7 ->	22 ->	23
0 ->	2 ->	15 ->	14 ->	27 ->	26 ->	11 ->	10
0 ->	13 ->	12 ->	25 ->	24 ->	9 ->	8 ->	21
links: 1	2	3	4	5	6	7	

$$L = \frac{4}{27}*1 + \frac{4}{27}*2 + \frac{4}{27}*3 + \frac{4}{27}*4 + \frac{4}{27}*5 + \frac{4}{27}*6 + \frac{3}{27}*7$$

$$L = 3.88889.$$

Using the formula directly with $n = 28$ and $h = m = 7$,

$$L = \frac{7}{28-1} [2(7) + 1] = 3.88889.$$

7.7.1. Analysis of Mean Path Length

Recall that IAT is the mean time in seconds between arrival of external messages to any arbitrary link in a spiral network, regardless of size. Therefore external messages arrive at the rate of 1/IAT. Since each node has four connected links (see chapter 5), the arrival rate of external messages to each node is 4/IAT. If we define Y to be the total number of messages per second entering the entire network, then $Y = 4n/IAT$, since there are n nodes in the network. The one way mean path length is therefore the ratio of the total average one way link traffic (λ), to the average one way offered load [45]. Letting $Y' = Y/2 = 2n/IAT$, the mean number of links traversed by a typical message is the ratio λ/Y'. But from the previous theorem, we have a closed form expression for the mean path length.

Our current purpose is to compare path length values obtained from three independent approaches: 1) the ratio λ/Y'; 2) the equation from theorem 6; and 3) the measured result from simulation.

Table 7.2 summarizes path length results found using these three independent approaches. The values of λ in the second column of table 7.2 are found by using theorem 5. $\gamma' = 2n/IAT$, where IAT = 20 seconds. The column L_r is found by using the ratio described above (λ/γ'). The 5th column (L_t) contains theoretical results from theorem 6. Note that the columns L_r and L_t are identical, as they must be if our analysis is accurate. The last column (L_m) contains the values found by the simulation model. In the model

$$L_m = \frac{\text{total number of hops for all messages}}{\text{number of messages}}.$$

Notice that without exception for the 12 networks analyzed, the measured result (L_m) is extremely close to theoretical expectations (L_r and L_t).

Table 7.2. Mean Path Length Comparison.

(in number of hops)

n	λ	γ'	L_r	L_t	L_m
16	3.8399	1.6	2.4000	2.4000	2.3923
20	5.7894	2.0	2.8947	2.8947	2.9005
28	10.8889	2.8	3.8889	3.8889	3.8772
32	14.0386	3.2	4.3871	4.3871	4.3488
40	21.5384	4.0	5.3846	5.3846	5.3597
44	25.8883	4.4	5.8837	5.8837	5.9078
52	35.7885	5.2	6.8824	6.8824	6.9240
56	41.3381	5.6	7.3818	7.3818	7.3497
64	53.6380	6.4	8.3809	8.3809	8.4295
68	60.3881	6.8	8.8806	8.8806	8.9185
76	75.0881	7.6	9.8800	9.8800	9.8270
80	83.0376	8.0	10.3797	10.3797	10.2005

7.8. Analysis of System Utilization

This section discusses the calculation and comparison of the mean link utilization (ρ), in any size spiral network, regardless of failures. We first find exact theoretical values (ρ_t) by using theorem 5 to find the total average one way link traffic (λ). These results are compared to a shorter approximation approach (ρ_a), and to actual measured values found from simulation.

Assuming a Poisson message arrival rate with λ messages per second arriving on the average, exponentially distributed message lengths with mean $1/\mu$, and infinitely large buffers, the following expressions can be used [45]:

Theoretical: $\rho_t = \frac{\lambda}{\mu C}$, $\frac{1}{\mu}$ = mean message length,

$C = \Sigma C_i$, total network capacity in bits per second, and

λ = total average one way link traffic.

Approximation (as defined in section 7.5):

$\rho_a = \frac{\lambda_i}{\mu C_i} L_m$, $\lambda_i = 1/IAT$,

C_i = link capacity in bits per second, and

L_m = mean path length.

Simulation model:

$$\rho_m = \frac{\text{average time all links busy}}{\text{total network time}}.$$

Table 7.3 summarizes these results for the 12 spiral networks used throughout our analysis. The values of λ used here are from table 7.2. Notice how closely the approximation (ρ_a) is to the theoretical utilization (ρ_t). This is the confirmation promised in section 7.5 on the goodness of ρ_a. The high confidence in our model is again confirmed as the values found from direct simulation, without exception, are extremely close to the theoretically expected utilization.

Table 7.3. System Utilization Comparison.

m	C	ρ_t	ρ_a	ρ_m
4	614400	.05000	.04984	.04820
5	768000	.06031	.06043	.05890
7	1075200	.08102	.08078	.07972
8	1228800	.09140	.09060	.08849
10	1536000	.11218	.11166	.11286
11	1689600	.12258	.12308	.12104
13	1996800	.14338	.14425	.14673
14	2150400	.15379	.15312	.15060
16	2457600	.17460	.17561	.17136
17	2611200	.18501	.18580	.17796
19	2918400	.20583	.20473	.20226
20	3072000	.21624	.21251	.21263

7.9. Analysis of Mean Queue Length

This section defines and compares the theoretical mean queue length to the values found using the simulation model. Let E(n) represent the mean queue length in any size spiral network. If we assume a Poisson message arrival rate with λ messages per second arriving on the average, exponentially distributed message lengths with mean $1/\mu$, and infinitely large buffers, we can use the results derived in [45] for the mean queue length computations:

1). Theoretical mean queue length based on theoretical utilization (ρ_t):

$$E_{tt}(n) = \frac{\rho_t}{1 - \rho_t}, \text{ where } \rho_t = \lambda/\mu C \text{ as calculated in table 7.3.}$$

2). Theoretical mean queue length based on simulation model utilization (ρ_m):

$$E_{tm}(n) = \frac{\rho_m}{1 - \rho_m}, \text{ where } \rho_m \text{ is the measured result from simulation as reported in table 7.3.}$$

3). Simulation model value calculated directly, independent of ρ_m calculations:

$$E_m(n) = \frac{\text{sum of the average of all queues}}{\text{number of messages}}.$$

In this last case, after each message is delivered, a snapshot of all queue lengths is averaged, and added to a running total. When the total number of messages used for statistics has been delivered, this sum of queue averages is divided by that total number of messages used for statistics. (See Appendix B for a discussion of the simulation model). Table 7.4 summarizes for comparison the mean queue length values found using the three approaches.

While the results for the mean queue length are consistently close for theoretical ($E_{tt}(n)$) and model ($E_{tm}(n)$) system utilization values, in every case the mean queue length results from simulation ($E_m(n)$) is consistently smaller. Notice in the table that as the size of the network increases, $E_m(n)$ approaches the theoretically expected values. $E_m(n)$ is much less than the theoretical expectations for smaller networks because of the common parameters selected for use in all networks. When the network contains only four modules (16 nodes). these common parameters result in the highest number of empty queues. The common link speed is 19200 bits per second, with messages arriving with frequency .05 messages per second. The mean queue lengths $E_m(n)$ were found by first including the empty queues in computations, and then by excluding these empty queues. Simulation results confirm that when empty queues are removed from computations, without exception, the values of $E_m(n)$ exceed theoretical expectation. When these empty queues are left in the computations, as the mean path length increases (because of an increase in network size), the number of empty queues decreases. This decrease occurs because there is a greater chance that a message must wait for the transmission link. The increase in mean path link explains why the values of $E_m(n)$ approach the theoretical expectations ($E_{tt}(n)$ and $E_{tm}(n)$) as the network gets larger.

Table 7.4. Mean Queue Length Comparison.

m	$E_{tt}(n)$	$E_{tm}(n)$	$E_m(n)$
4	.05263	.05064	.00483
5	.06418	.06259	.00947
7	.08816	.08663	.01909
8	.10059	.09708	.02436
10	.12635	.12722	.04999
11	.13971	.13771	.06076
13	.16738	.17196	.10231
14	.18174	.17730	.11078
16	.21153	.20680	.15505
17	.22701	.21649	.17986
19	.25918	.25354	.22606
20	.27592	.27005	.26373

7.10. Analysis of Mean Network Delay

We are now ready to calculate and compare the time that it takes for any size spiral network to deliver a message. Assuming Poisson message arrivals to an arbitrary link (i) with rate λ messages/second, exponentially distributed messages with mean length $1/\mu$, and infinitely large buffer capacity, the average delay in seconds incurred by messages at the (i)th link is [45]:

$$T_i = \frac{1}{\mu_i C_i - \lambda_i}. \qquad (7.10.1)$$

This result invokes Kleinrock's assumption [46] that the operation of individual nodes is independent of each other. This assumption is approximately valid when the overall system utilization is less than .5000. The highest utilization of any of the 12 spiral networks simulated and analyzed in our research was .21263 (see table 7.3 for confirmation).

Based on equation 7.10.1, the total overall average one way delay for any size spiral network is defined to be [45]:

$$T = \frac{1}{\gamma'} \sum_{i} \lambda_i T_i, \qquad (7.10.2)$$

where γ' is the average one way offered traffic load. In our case, since we have three types of links (Types I, II, and III), the overall delay equation T can be modified to reflect these types:

$$T = \frac{1}{\gamma'}\left[\sum_{i=1}^{n/2} \lambda_1 T_I + \sum_{i=1}^{n/2} \lambda_2 T_{II} + \sum_{i=1}^{n} \lambda_3 T_{III}\right], \text{ or}$$

(7.10.3) $$T = \frac{1}{\gamma'}\left[\frac{n}{2}(\lambda_1 T_I + \lambda_2 T_{II}) + n\lambda_3 T_{III}\right],$$

where λ_1, λ_2, and λ_3 are as defined in section 7.6.2, and

$$T_I = \frac{1}{\mu_i C_i - \lambda_1},$$

$$T_{II} = \frac{1}{\mu_i C_i - \lambda_2},$$

$$T_{III} = \frac{1}{\mu_i C_i - \lambda_3}.$$

Table 7.5 shows the theoretical results (in seconds) of applying equation 7.10.3. These values (in the column labeled T_t) are compared to the delay values found in the simulation model (T_m). In the model, the actual delay for each message is calculated and added to a running total. When all messages used for statistics have been collected, this delay total is divided by the number of messages used. (See Appendix B for a discussion of the simulation model).

Table 7.5. Mean Network Delay Comparison.

(in seconds)

m	T_t	T_m
4	1.07223	1.06421
5	1.31367	1.37995
7	1.85592	1.96620
8	2.14482	2.26595
10	2.76750	3.19398
11	3.10376	3.66591
13	3.83463	4.94012
14	4.23261	5.28370
16	5.10178	6.63168
17	5.57783	7.40714
19	6.62851	8.68546
20	7.20997	9.52298

8. ANALYSIS OF SPIRAL NETWORK TOPOLOGY UNDER FAILURES

8.1. Introduction

Failures analyzed in this section range from a single arbitrary node, to catastrophic conditions where complete modules have failed. Conclusions drawn are based on two approaches to analysis. 1) Simulation results were analyzed and compared to theoretically expected values; and 2) The spiral network threading pattern was thoroughly analyzed. The first part of this chapter addresses arbitrary node failures. The last part is the result of analyzing the network threading pattern to determine the impact failed modules have on performance. Appendix D contains the Summary of Simulation Results used to draw conclusions based on model analysis.

We conclude that failure of any arbitrary single node (or a few arbitrary nodes), has minimum impact on the operation of remaining good nodes. Even catastrophic failure patterns possible through intentional human intervention, may not completely disconnect the network. Although the 7-module, 28-node spiral network is used throughout, results apply for any size spiral network. The spiral network expansion algorithm subsumes the 7-module network as a larger network is built, thereby causing the results found for 7-modules to apply to any larger spiral network.

8.2. Arbitrary Node Failure(s)

Failure of any arbitrary single node, or a few arbitrary nodes in a spiral network, has minimum impact on the network operations. As expected, the percentage of undelivered messages due to this single failure is proportional to the size of the analyzed network. The larger the network, the less impact a single, or few failures have on overall network performance.

Figures 8-1 through 8-3 graphically display the impact of zero, one, two or three failures on 12 different size spiral networks. All networks were run using the same input parameters. The common link speed was 19200, 4000 messages were delivered to reach simulated steady state, and 4000 messages were used to gather statistics. Message arrivals followed a Poisson rate with mean $\lambda = .05$ messages per second, and the message sizes were selected from an exponential distribution with mean $1/\mu = 1000$ 8-bit characters. Appendix B contains a detailed description of the simulation model. And complete simulation summary statistics are in Appendix D.

As expected, the impact of failures on mean queue length, overall delay, and utilization is inversely proportional to

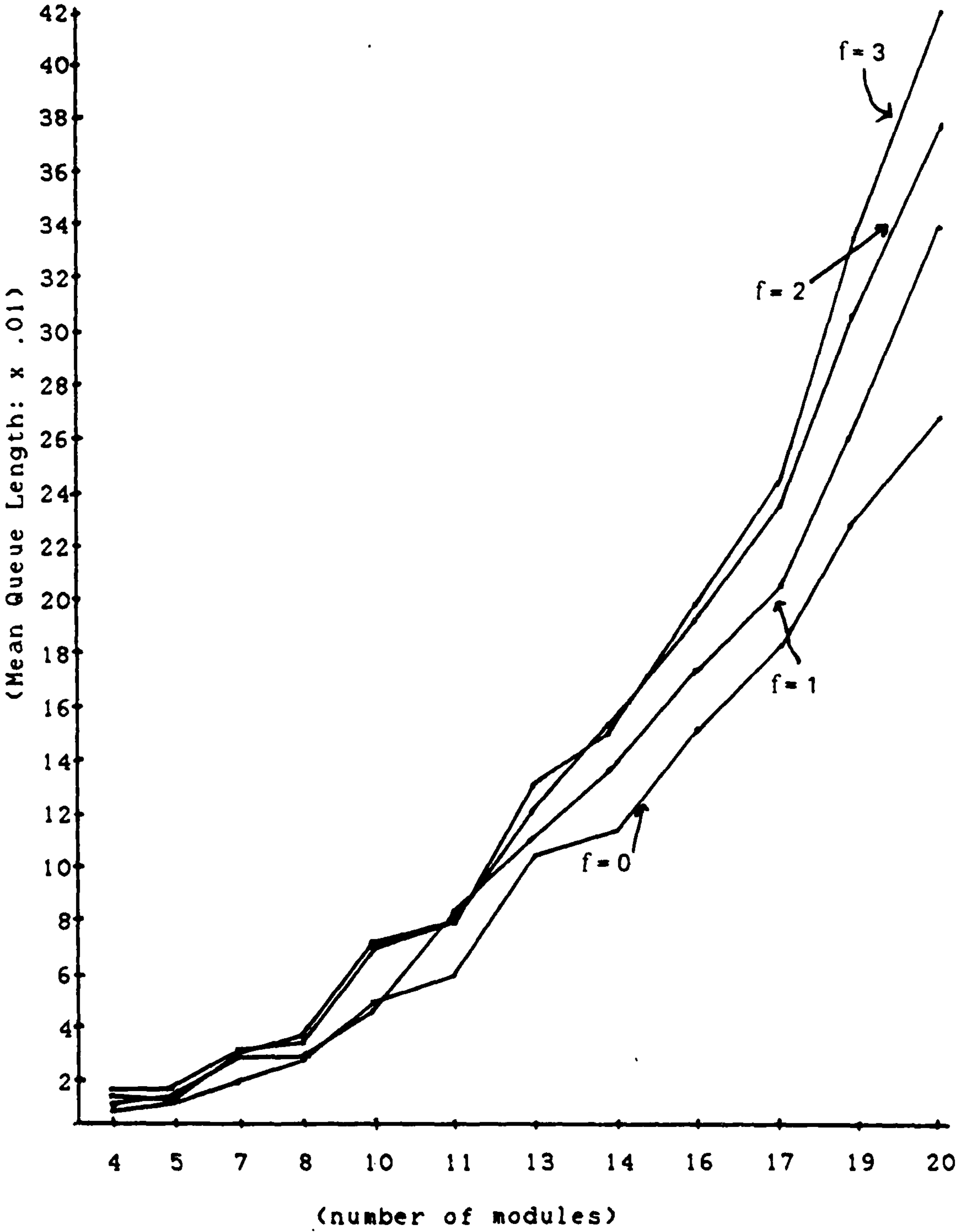


Figure 8-1. Mean Queue Length Comparison.

(failures (f) = 0, 1, 2, 3)

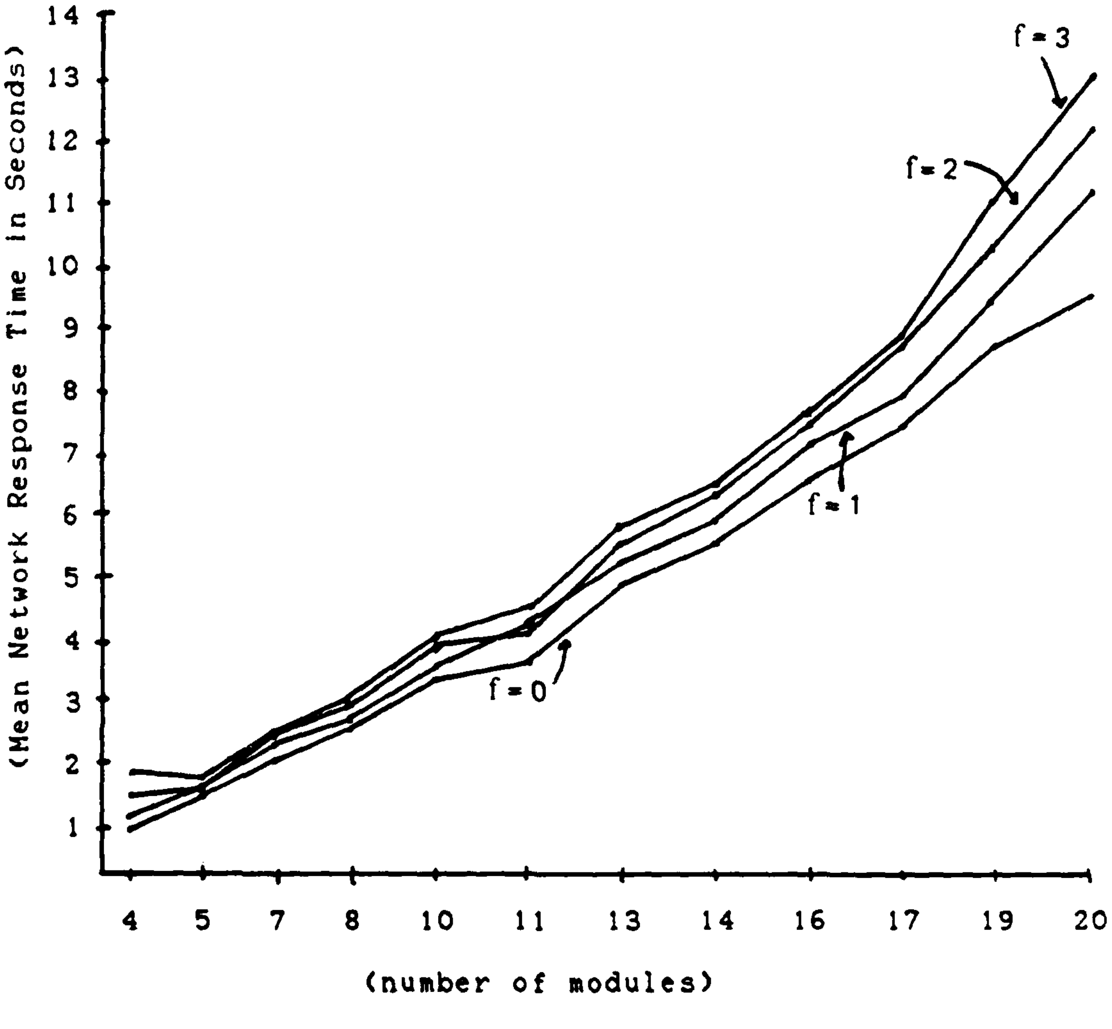


Figure 8-2. Network Response Time Comparison (in seconds).

(failures (f) = 0, 1, 2, 3)

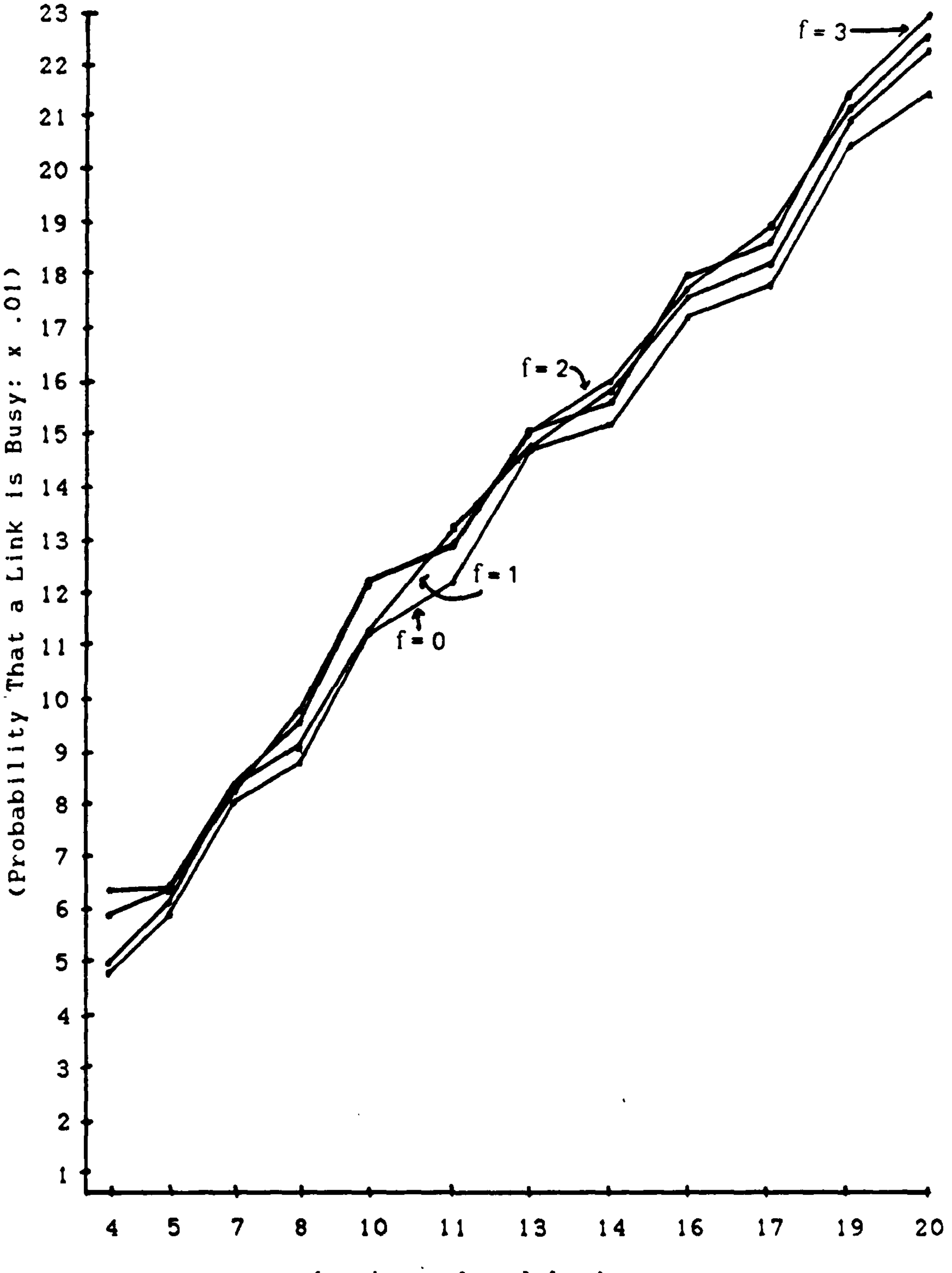


Figure 8-3. System Utilization Comparison.

(failures (f) = 0, 1, 2, 3)

the size of the network. Of key significance is how close the four curves are for each of the attributes compared. This closeness reflects how little an additional node failure impacts overall network performance. Statistics used to plot the curves in figures 8-1 through 8-3 are summarized in table form in Appendix C.

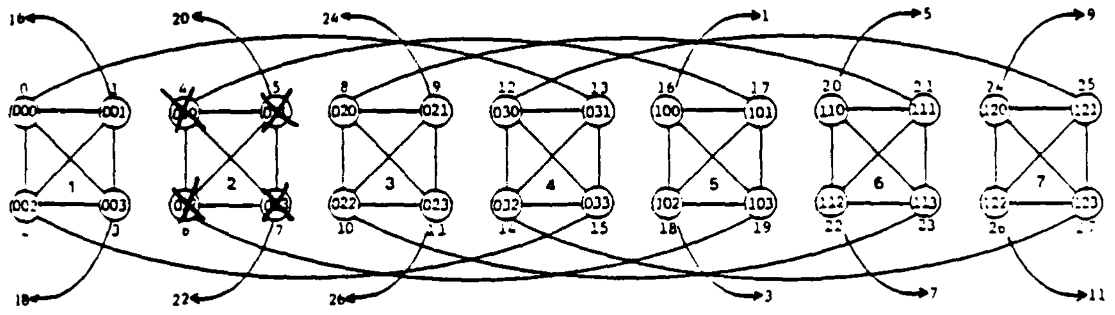
Table 8.1 shows how closely the actual percentage of undelivered messages due to failures, as calculated in the simulation model, parallel the theoretical expectations (f_t). To arrive at the measured percentage, the simulation program simply counted the number of undelivered messages, and then divided that result by the total number of messages used for statistics (4000). The theoretical expectation (f_t) is the percentage of failed nodes in each size spiral network.

Table 8.1. Comparison of Undelivered Messages.

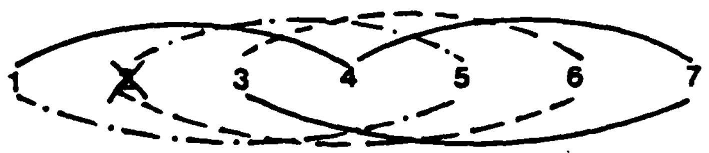
	Percentage of Failures					
m	1	f_t	2	f_t	3	f_t
4	6.31	6.25	12.77	12.50	19.21	18.75
5	4.85	5.00	10.37	10.00	15.45	15.00
7	3.86	3.57	7.66	7.14	10.94	10.71
8	3.31	3.13	6.51	6.25	9.65	9.38
10	2.59	2.50	5.24	5.00	7.68	7.50
11	2.24	2.27	4.30	4.55	6.62	6.82
13	1.85	1.92	4.07	3.85	5.41	5.77
14	1.85	1.79	3.71	3.57	5.32	5.36
16	1.62	1.56	3.14	3.13	4.55	4.69
17	1.21	1.47	2.89	2.94	4.65	4.41
19	1.32	1.32	2.64	2.63	3.85	3.95
20	1.30	1.25	2.33	2.50	3.76	3.75

8.3. Failure of a Complete Module

Any single module loss still leaves two disjoint paths to all other modules. For example, consider the 7-module network repeated in figure 8-4. If module #2 is destroyed, the two paths connecting the remaining nodes exist along the top and bottom spirals. Further, the threading pattern for this failure case (figure 8-4(b)) quickly confirms that all remaining modules are still connected.



a). Seven Module Spiral Network With One Module Killed.



b). Seven Module Network Threading Pattern.

Figure 8-4. Complete Module Failure.

8.4. Failure of Multiple Modules

8.4.1. Two Adjacent Modules

The failure of any two physically adjacent modules, and an additional one two modules away, is no worse than the loss of the two adjacent modules. Further, there are still two disjoint paths to all remaining good connected modules. For example, again consider the 7-module network's threading pattern. Shown in figure 8-5 is the case where modules 6 and 7 have failed. Notice that modules 1, 2, 4, and 5 are still connected along top and bottom spirals. Notice also that module #3 is already isolated from the four other good modules. The four nodes comprising module 3 can talk to each other, but not to anyone else. So if module #3 also should fail, there is no additional impact on modules 1, 2, 4, and 5.

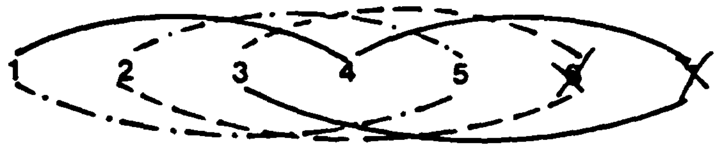


Figure 8-5. Threading Pattern With Two Adjacent Modules Killed.

8.4.2. Every Other Even/Odd Module Lost

The loss of complete network connectivity among modules is extremely difficult. For a spiral network containing seven or more modules, even if every even (or odd) module within the network fails, there is still connectivity between at least two modules. For the 7-module spiral network, modules 1, 3, 5, and 7 failing still leaves modules 2 and 6 connected. If 2, 4, and 6 failed, the module pairs 1 and 5, and 3 and 7 remain connected. The module threading patterns shown in figures 8-6 and 8-7 are graphic depictions of these cases. In both cases, connected modules are reachable along both top and bottom spirals. In figure 8-6, the isolated nodes on module #4 can still communicate with each other.

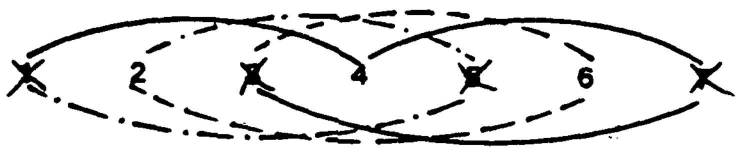


Figure 8-6. Every Odd Numbered Module Killed.

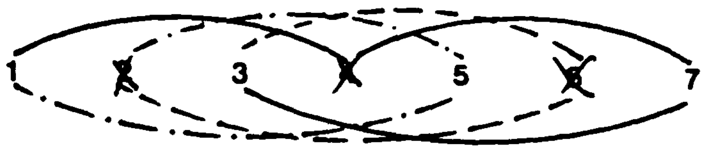


Figure 8-7. Every Even Numbered Module Killed.

8.4.3. Catastrophic Node and Module Failure

If at least one link exists to any module, then all nodes that can reach that link, can also gain access to the distant module(s). For example, consider the severe failure pattern of figure 8-8. The link between nodes 3 and 18 can be used by the remaining good nodes to communicate among themselves. In this unique case, modules 2, 3, 6, and 7, have failed, plus select nodes on other modules. Even in this catastrophic case, there is still connectivity among good nodes.

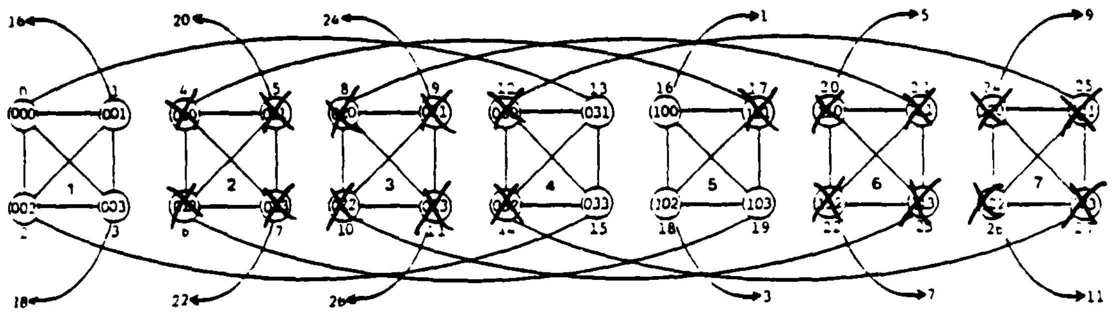


Figure 8-8. Several Modules and Nodes Killed.

8.4.4. Failure of Half of Each Module

Failure of the right (or left) half of every even (odd) module, still leaves access to a partitioned set of good modules and nodes. Further, these connected good modules still have paths along top and bottom spirals. In the 7-module case shown in figure 8-9, the right sides of each even module is destroyed. Notice that modules 1, 2, and 5 are still connected along top and bottom spirals. This is also true of modules 3, 4, and 7. Further, if failures are due to complete module(s) loss, as long as there is connectivity between at least two modules, there will be a minimum of two disjoint paths connecting these modules.

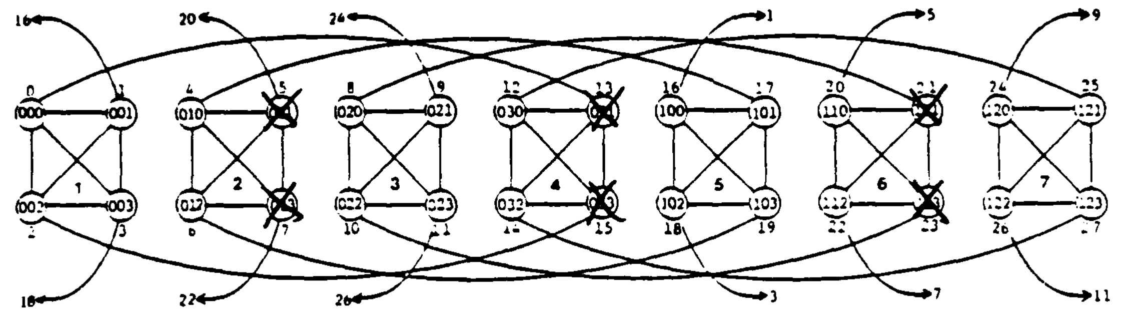


Figure 8-9. Half of Each Even Module Killed.

8.5. Summary

It should be apparent from the above discussion that significant network performance impact is felt only after more than a few arbitrary nodes have failed. Even in catastrophic cases, connected good nodes still communicate, and therefore the spiral architecture tolerate failure(s) extremely well. Although certain failure patterns cause complete direction change, and thus increase the message delay, the significance is that messages still reach their destination even with the failures. Some of the failure combinations analyzed will not generally occur as a result of randomness. These patterns were selected to demonstrate spiral's extreme tolerance to failures.

As the number of failures increase, a point is reached where the delay becomes shorter than in the error free case. This situation results when so many nodes and modules have failed that path lengths between remaining good nodes is shortened considerably.

Obviously certain failure patterns are likely only as a result of deliberate network sabotage. Even in this highly unlikely case, every pair of link nodes in the network must fail simultaneously to completely disconnect all modules. But even if every pair of link nodes did fail, nodes comprising a module still have local connectivity, and therefore, the ability to communicate. In the absence of a sabotuer, the spiral topology displays an extremely high tolerance to node and link failures. For sure, the fault tolerance of the spiral topology exceeds that of the traditional topologies discussed in chapter 3, and at a small expense for duplication of hardware.

9. CONCLUSIONS

The six most important direct consequences of the spiral computer network architecture are the spiral topology's 1) ease of expansion; 2) fast, on-the-fly self-routing; 3) extremely high tolerance to faults; 4) increased network security; 5) potential for the total elimination of store and forward transmission due to routing decision delays; and 6) rendering the maximum path length issue moot.

Based on our thorough analysis, we conclude unequivocally that the spiral topology is a major contribution to the discipline of Computer Communications. Results in chapters 5 through 8 confirm that the spiral architecture is indeed easily expandable, highly fault tolerant, and self-routing. Also confirmed is the topology's applicability to any general network environment. The architecture can be use to connect computer nodes to form local, metropolitan, and wide area networks. If used, this topology should prove to be a major advantage to the telecommunications industry as that industry throughout the world continues to evolve towards a global Integrated Services Digital Network.

Finally, the fast, on-the-fly routing attribute affords a tremendous opportunity to expand the use of fiber optics technology in local area computer networks.

10. TOPICS FOR ADDITIONAL STUDY

The sequential listing of the following recommended topics for additional study in no way suggests a prioritized ordering. The only purpose in enumerating these topics is for organization.

1. Link Versus Node Failure

The results reported in this research were based on the analysis of various size spiral networks, with and without failed nodes. When a node failed, it automatically destroyed all four of the directly connected links. However, it is possible for a single or set of links to fail, and still leave a node accessible. If individual links were allowed to fail, and the resulting topology analyzed, the spiral routing algorithm should still operate as designed. Intuitively, one would expect the impact of link failures on performance to be less severe than when nodes fail. Analyzing the spiral topology for arbitrary link failures should prove to be challenging.

2. Destination Address Variability

Mathematically, a study of the variability in the number of messages sent to each node in an error free spiral network may prove interesting. Does the variability increase directly proportional to the network size, or is it fairly independent of size? Perhaps at steady state, the variability in the number of messages sent to each node remains constant, regardless of the size of the network.

3. Variability in Number of Empty Queues

Again mathematically, how does the number of empty queues vary with network size in an error free spiral network? We saw in chapter 8 that as the network size grew, the measured mean queue length approached the theoretically expected results from the lower side. Do these two approaches converge because of a uniform increase in queue lengths throughout the network, or is the increase mainly limited to a certain type of link (Type I, II, or III)?

4. Circuit and Packet Switching

Of interest and significance to a potential Integrated Services Digital Network user contemplating the use of the spiral architecture, is the study of the performance of the spiral network architecture using circuit switching only, and a mix of circuit and packet switching. For example, the larger the network, the longer is the mean path for packet switching. Does this also imply that the mean circuit switched path increases with network size? Or will the probability of blocking in a circuit switched environment be reduced because on the average, a larger portion of the connections will be short, and perhaps avoid contact with the occasional "long" connection? Further, what about integrating circuit and packet switching technology on the same common channel? Intuitively, the 2B + D CCITT standard discussed in chapter 2 could be implemented using the spiral architecture. Then, the packet traffic would not have to compete with circuit switching for use of the links. Rather, packet and circuit switched traffic would share the channels. What would performance be like in this situation? Can a set of balance curves be derived that allows one to achieve a better balance between circuit and packet switching, whether the traffic is integrated in some sort of 2B + D scheme, or competing for the network resources?

5. Closed Form Expressions When Failures Occur

Are reasonable and meaningful results possible in closed form for failed nodes, similar to those reported in chapter 7 for error free spiral networks? If the analysis based on individual link (versus node) failures is done, are similar closed form expressions possible?

6. Alternate Approaches to Modeling the Spiral Topology

The current simulation model for the spiral topology is mostly matrix driven, with global variables, and parameter passing among program subroutines. The C programming language contains constructs that can be used and passed as a generic shell. Would the use of the generic shell yield a more efficient, or "better" simulation model? Is the use of huge matrices (i. e. 256 x 19, 1024 x 40, 1024 x 60) a standard and recommended approach to developing a simulation model? In the 1024 x 60 matrix used to simulate the spiral architecture nodes, 51 of the columns in each row were used as queue slots for each of the four links at each of (n) nodes. Is it possible to maintain a common "pool" of queue slots that are allocated and return dynamically? These are issues of (some) interest to Computer Scientist.

7. Spiral Topology Applied to Computer Hardware Memories

We see no immediate reason why the highly fault tolerant, fast, on-the-fly routing attributes of the spiral topology cannot be used as an interconnection network for connecting computer hardware memories. Perhaps the spiral architecture could also be used for fast retrieval from shared memory units.

8. Behavior as Steady State is Approached

There are mathematicians interested in the behavior of systems during the transition period leading up to steady state. Treating the spiral topology as such a system, one may study several issues. How does the number of messages sent to each node vary? What happens to mean and maximum queue lengths? What about the behavior of link utilization? Is steady state approached uniformly, or do certain types of links (Type I, II, or III) approach steady state faster than others? What about the variability in message sizes?

9. Upper Bound on Maximum Number of Paths

We reported in theorem 1 on the number of totally disjoint paths in any size spiral network without failures. Using combinatorics, is it possible to place an upper bound on the total number of paths, where links are shared? Can this be done in closed form allowing for failed nodes?

11. LIST OF REFERENCES

1. C. M. Verber, B. J. Brownstein, R. P. Kenan, Annual Report on Definition Study for Intelligent Optical Nodes for Computer Networks, pp. 29.1.2-29.1.7, 1981.

2. Oliver C. Ibe and David T. Gibson, "Protocols for Integrated Voice and Data Local Area Networks", IEEE Communications Magazine, pp. 30-36, July 1986.

3. Mischa Schwartz, Telecommunication Networks Protocols, Modeling and Analysis, pp. 661-715, 1987, Addison-Wesley, MA.

4. Thomas J. Herr and Thomas J. Plevyak, "ISDN: The Opportunity Begins", IEEE Communications Magazine, pp. 6-10, November 1986.

5. Mario Gerla and Rodolfo A. Pazos-Rangel, "Bandwidth Allocation and Routing in ISDN's", IEEE Communications Magazine, pp. 16-26, February 1984.

6. G. S. Bhusri, "Considerations for ISDN Planning and Implementation", IEEE Communications Magazine, pp. 18-32, January 1984.

7. W. Victor Tang, "ISDN - New Vistas in Information

Processing", IEEE Communications Magazine, pp. 11-16, November 1986.

8. Dwight B. Davis, "Phone Companies Argue Over New Standards", High Technology Magazine, pp. 26-31, August 1987.

9. A. R. K. Sastry, "Performance Objectives for ISDN's", IEEE Communications Magazine, pp. 49-54, January 1984.

10. C. S. Skrzypczak, J. H. Weber, and W. Falconer, "Bell System Planning of ISDN", IEEE International Conference on Communications, pp. 19.6.1-19.6.6, 1981.

11. R. M. Wienski, "Evolution to ISDN Within the Bell Operating Companies", IEEE Communications Magazine, pp. 33-41, January 1984.

12. D. J. Kostas, "Transition to ISDN - An Overview", IEEE Communications Magazine, pp. 11-17, January 1984.

13. Irwin Dorros, "Keynote Address, The ISDN - A challenge and Opportunity for the '80's", IEEE International Conference on Communications, pp. 17.0.1-17.0.5, 1981.

14. Irwin Dorros, "ISDN", IEEE Communications Magazine, pp. 16-19, March 1981.

15. Wayne J. Felts, Warren Gifford, and Frank J. Gratzer, "Bell's Concept of the ISDN", Telephony, pp. 43-51, October 1982.

16. Cliff Hoppitt, "ISDN Evolution: From Copper to Fiber in Easy Stages", IEEE Communications Magazine, pp. 17-22, November 1986.

17. F. T. Andrews, Jr., "ISDN '83", IEEE Communications Magazine, pp. 6-10, January 1984.

18. W. S. Gifford, "ISDN User-Network Interfaces", IEEE Journal on Selected Areas in Communications, pp. 343-348, May 1986.

19. T. Irmer, "An Idea Turns Into Reality - CCITT Activities on the Way to ISDN", IEEE Journal on Selected Areas in Communication, pp. 316-319, May 1986.

20. "Integrated Services Digital Network: Technology and Implementations-II", special issue, IEEE Journal on Selected Areas in Communications, November 1986.

21. C. R. Williamson, "Opening the Digital Pipe, Bell System Overview", IEEE International Conference on Communications, pp. 29.1.1-29.1.7, 1981.

22. M. J. Ferguson, "Computation of the Variance of the Waiting Time for Token Rings", IEEE Journal on Selected Areas in Communications, pp. 775-782, September 1986.

23. L. C. Mitchell and D. A. Lide, "End-to-End Performance Modeling of Local Area Networks", IEEE Journal on Selected Areas in Communications, pp.975-985, September 1986.

24. P. J. B. King and I. Mitrani, "Modeling a Slotted Ring Local Area Network", IEEE Transactions on Computers, pp. 554-561, May 1987.

25. Werner Bux, "Local-Area Subnetworks: A Performance Comparison", IEEE Transactions on Communications, pp. 1465-1473, October 1981.

26. Mischa Schwartz, Telecommunication Networks Protocols, Modeling and Analysis, pp. 451-464, 1987, Addison-Wesley, MA.

27. Simon S. Lam, "A Carrier Sense Multiple Access Protocol for Local Networks", Computer Networks, pp. 21-32, 1980.

28. Leonard Kleinrock and Fouad A. Tobagi, "Packet Switching in Radio Channels: Part I-Carrier Sense Multiple-Access Modes and Their Throughput-Delay Characteristics", IEEE Transactions on Communications,

pp. 1400-1416, December 1975.

29. R. M. Metcalfe and D.R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", Communications of the ACM, July 1976.

30. A. West and A. Davison, "CNET-A Cheap Network for Distributed Computing", Department of Computer Science and Statistics, Queen Mary College, University of London, Report TR 120, March 1978.

31. K. J. Biba and J. W. Yeh, "FordNet: A Front-End Approach to Local Computer Networks", Proc. Local Area Communications Network Symposium, Boston, MA, May 1979.

32. Luis F. M. DeMoraes and Izhak Rubin, "Analysis and Comparison of Message Queueing Delays in Token-Rings and Token-Buses Local Area Networks", International Conference on Communications, pp. 130-134, 1984.

33. P. J. Kuehn, "Multiqueue Systems with Nonexhaustive Cyclic Service", Bell Systems Technical Journal, Vol 58, pp. 671-698, 1979.

34. W. Chou, Computer Communications, Vol. 1, chapter 10, 1983, Prentice Hall Inc, N. J.; 1983.

35. W. Chou, "Terminal Response Time on Polled Teleprocessing Networks", Computer Networking Symposium, December 1978.

36. A. E. Kamal, "Star Local Area Networks: A Performance Study", IEEE Transactions on Computers, pp. 483-499, April 1987.

37. Anthony S. Acampora and Michael G. Hluchyj, "A New Local Area Network Architecture Using a Centralized Bus", IEEE Communications Magazine, pp. 12-21, August 1984.

38. A. S. Acampora, M.G. Hluchyj, and C.D. Tsao, "A Centralized-Bus Architecture for Local Area Networks", Proceedings of the International Conference on Comm., pp. 932-938, 1983.

39. A. G. Fraser, "DATAKIT - A Modular Network for Synchronous and Asynchronous Traffic", Proceedings of the International Conference on Communications, pp. 20.2.1 - 20.2.3, June 1979.

40. William Stallings, "Local Network Performance", IEEE Communications Magazine, pp. 27-36, February 1984.

41. Bart W. Stuck, "Calculating the Maximum Mean Data Rate in Local Area Networks", IEEE Computer Magazine, pp. 72-75,

May 1983.

42. Mischa Schwartz, Computer-Communication Network Design and Analysis, pp. 242-253, 1977, Prentice Hall, N. J.

43. "Interworking With Telephone Network - First Draft Recommendation", CCITT Temporary Document No. 616-E, Working Party XI/6, Geneva, November 30-December 3, 1982.

44. "Interworking Between ISDN and the Analogue Telephone Network for Data Communication", CCITT Comm. XVIII No. JR, ISDN Experts Meeting, Kyoto, Japan, February 14-25, 1983.

45. Mischa Schwartz, Computer-Communication Network Design and Analysis, pp. 71-115, 1977, Prentice Hall, N.J.

46. Leonard Kleinrock, Queueing Systems, 1, 1975; 2, 1976, Wiley-Interscience, N. Y.

47. O. M. M. Mitchell, "Implementing ISDN in the United States", IEEE Journal on Selected Areas in Communications, pp. 398-406, May 1986.

48. "Information Network Architecture Evaluation and Development", Draft Report, Research Triangle Institute, N. C., pp. 22 - 25, December 1986.

A. EVOLUTION TO ISDN WITHIN THE BELL OPERATING COMPANIES

The Bell System evolution toward ISDN started in 1962 with the introduction of the T1 carrier system. The Bell System has passed many additional milestones since then, including the following [10]:

* 1965 - Stored Program Controlled Switching
* 1974 - Digital Data System/Dataphone Digital Service
* 1976 - Time Division Tandem Switch
* 1976 - Packet Switched signaling
* 1981 - Time division Local Switch

The Bell Operating Companies' (BOC's) telecommunications networks today are primarily 4-kHz voice networks. The origins of technologies that enable the economic evolution toward an ISDN, however, began over twenty years ago with the introduction of the T-carrier system and the AT&T Western Electric 1ESS switch [47].

The T-carrier system provides a 1.544-Mbps facility carrying 24 to 64-kbps channels plus framing bits. Some of the bits in the 64-kbps channels are robbed for signaling purposes. While this is not a perceptible degradation on voice circuits, it would cause unacceptable error rates for data applications. Hence, as on Dataphone Digital Service (DDS), only 56-kbps can be provided for customer applications.

ISDN evolution strategies range from overlay strategies, in which a parallel ISDN network is deployed side-by-side with the existing voice network, to replacement strategies, in which geographic areas undergo replacement of existing equipment with ISDN equipment. Given the large capital investment in the telecommunications networks of the BOC's, a strategy which combines the two is appropriate. Existing equipment must be augmented wherever possible, with new equipment deployed when required [11].

The telecommunications networks of the BOC's will evolve to ISDN in basically a four-step fashion [11]. First, new transition services will be introduced which offer ISDN-like services. These services will be low in development and deployment cost so as to minimize the inherent risk associated with new service offerings. Examples include Circuit Switched Digital Capability and Local Area Data Transport. Second, as the existing network grows to meet the rising demands of customer traffic, new ISDN compatible equipment will be deployed, seeding the network with facilities capable of meeting the ISDN standards. Examples of such equipment include fiber optics, 64-kbps clear channel transmission equipment and common channel signaling capabilities. Third, true ISDN service offerings will appear in areas where customer needs dictate them. Typically these will be business areas, usually located in downtown metropolitan areas. And fourth, as ISDN demand increases, ISDN capabilities will permeate the entire BOC network.

The existing public telecommunications network can be characterized as having five major components: the local loop, the local switch, the metro or interoffice facility, the tandem switch and the intercity facility. In addition to these, signaling is the "glue" that holds this entire process together [10].

J. Weber and C. Skrzypczak [10] and R. Wienski [11] discuss several areas in which digital progress must continue to evolve if we are to realize the ISDN as described by the CCITT. These areas are as follows:

1. ISDN Access Evolution
2. Local Loop Evolution
3. Metro/Interoffice Evolution
4. Tandem Switching Evolution
5. Intercity Facilities Evolution
6. Signaling Network Evolution
7. Interworking of ISDN with existing Services

The following paragraphs describe briefly underlying economic trends which impact the five major components of the public network and the seven areas of the ISDN evolution [10, 11].

A.1. ISDN Access Evolution

Several access methods are under study to provide data and voice over the same 2-wire loop [15]. The first technique uses time compression multiplexing (TCM) to provide a 56-kbps digital data transmission channel. The second technique, the digital subscriber line (DSL), can be provided using TCM or other techniques to support multiple digital channels in the ISDN. The third technique, digital over analog, uses channel equipment to put data (up to 8-kbps) in the frequency spectrum above voice. For higher capacity applications, either multiple channels multiplexed together or broadband channels can be employed. The broadband channels can be provided using T-carrier, radio, lightwave or other systems.

A.2. Local Loop Evolution

The local loop is often regarded, along with the intercity facilities, as a bottleneck in providing digital capabilities [5]. This is true given the percentage of loops currently carrying digital signals. A key issue is the digital techniques that can be overlayed on the existing metallic loop plant. Four alternatives include four-wire baseband, multiplexing, low bit rate data above voice, and digital subscriber line [5].

A.3. Metro/Interoffice Evolution

Digital technology first penetrated the public telecommunication network in the metro facilities component with the introduction of T1 carriers in 1962. Since the public telecommunications network was originally an analog network, it was necessary to go through an analog to digital and digital to analog conversion each time a digital component was inserted. However, as digital technology began to proliferate, a new digital component was often interfaced directly with another digital rather than an analog component. This eliminated the need for conversion and further reduced the cost of the digital alternative [10].

A.4. Tandem Switch Evolution

The obvious choice for a tandem switching vehicle to support end-to-end digital connectivity is a stored program controlled (SPC) time division switch [see 10]. An SPC space division switch with relatively minor modifications can also support end-to-end digital connectivity. In the evolution to ISDN, both time and space division technologies will probably be employed.

A.5. Intercity Facility Evolution

In the intercity facility portion of the network, digital technology has not generally proven itself over the analog alternative based on economics alone [10]. The greater long haul breakthrough might occur with subrate (less than 56/64-kbps) voice.

A.6. Signaling Network Evolution

The signaling capability which ties together the five basic components of the public telecommunications network can be separated into two major segments [10]: Signaling from the customer premises to the line side of the local switch and signaling from the trunk side of the originating local switch to the trunk side of the terminating local switch. On the line side of the local switch and the loop, some of the future ISDN technologies such as DSL, will support the signaling needs of the ISDN. The Bell System began to introduce out-of-band common channel signaling in the portion of the network between central offices in 1976. It provides significant trunk efficiencies and faster call set up times. It is now proving to be capable of supporting extended routing, enabling the provision of a wide range of new and expanded service capabilities compatible with the evolution to ISDN.

A.7. Interworking of ISDN with Existing Services

Interworking with existing services will allow for a successful early deployment of ISDN in a smooth, step-by-step upward compatible fashion. Several schemes have been proposed to provide interworking [10, 43, 44].

B. SIMULATON MODEL DESCRIPTION

B.1. Introduction

The simulation model used in this research contains approximately 2700 lines of computer program source code. The C programming language was used, and the compiled source code runs on a computer system using the UNIX operating system. Most of the model development work was done on the VAX 11/780 computer system at North Carolina State University. To obtain the summary results used for comparative analysis, the completed model was run on a Gould computer system at the U. S. Air Force Academy in Colorado. The summaries of these simulation results are found in Appendix D. And the complete C source code listing is in Appendix E.

A matrix driven software development strategy was used to develop the final model that implements the spiral computer network topology concept. The actual building of a spiral network, with or without failed nodes, is implemented by using a matrix with 256 rows and 19 columns. The 256 rows allow a spiral network to contain at most 64 modules (256 nodes). See the declarations section in the documented source code listing in Appendix E for an explanation of how each column is used.

A matrix with 1024 rows and 40 columns contains information needed to keep track of, and send messages over the network. At most 1024 active messages can exist in the network at any one time. This limitation is easily changed to allow for more messages. However, we were able to deliver 8000 messages with mean arrival rate of .05 messages per second, and mean size of 1000 8-bit characters, over a 20 module (80 node) network without exceeding the 1024 active message limitation. See the declarations in the C program source code in Appendix E for an explanation of how the 40 columns are used.

The simulation program was written in two major phases. Phase I builds a spiral network and sets failures as desired. Phase II simulates the passing of messages over the network. The remainder of this appendix addresses these two phases, and summarizes the specific common parameters used to generate results that were compared to analytical ones.

B.2. Phase I: Construction of a Spiral Network

An individual running this program begins by responding to a series of computer generated prompts. The prompts include:

1. How many modules are desired?
2. Does the user want nodes or complete modules to fail?
3. Does the user want to save or see a copy of the current

network connectivity matrix?

The program checks to insure that a legal number of modules has been requested. For example, if the user requested more modules than there are node numbers available, then he is warned of this fact, and afforded the opportunity to add additional node numbers. If the number of requested modules is an integer multiple of 3 (see chapter 5), an explanation is returned explaining why that request is not valid. Once the desired spiral network configuration is obtained, the program begins Phase II with another series of prompts.

B.3. Phase II: Network Operation

This portion of the simulation experience also begins by having the user respond to a set of questions, which include:

1. Is the user ready to send traffic over the network?
2. What common link speed would the user like?
3. What is the mean message size (selected from an exponentially distributed population)?
4. What is the mean message interarrival time (selected from a Poisson distribution)?
5. How many messages should be delivered prior to gathering statistics?
6. How many messages should be used for statistics?

Once all necessary parameters have been entered, the program proceeds to simulate the network operation. Nodes send to each other with equal probability. Of course if nodes have failed, they neither generate, nor receive traffic. Since source nodes have no knowledge of the status of destination nodes, a short message may be generated notifying the sender of the failure. This message may be sent by the directly connected node that attempted delivery to the failed node.

Several statistical values are accumulated during the run, and others are found after all messages have been delivered. Every individual message is accounted for by number, and a complete route trace for each message is available. The Summary of Simulation Results in Appendix D shows exactly which statistics are reported. The simulation model source code contains several subroutines that were used to debug the code, and verify operations of key functions.

B.4. Specific Parameters Used for Comparative Analysis

The following parameters were used to generate the results reported on and compared in chapters 6, 7, and 8:

1. Number of modules: 4 through 20 (16 - 80 nodes), omitting integer multiples of 3 (see chapter 5 for explanation).
2. Failed nodes: node 1 only, 1 and 6, and 1, 6, and 11 for each of the networks containing 4 through 20 modules.

3. Mean message size: 1000 8-bit characters

4. Population distribution for message size: exponential.

5. Speed, all links: 19200 bits per second.

6. Mean message interarrival time: 20 seconds.

7. Message arrival pattern: Poisson.

8. Total messages delivered to reach steady state: 4000.

9. Total messages used for statistics: 4000.

C.1. RESPONS, TIME SUMMARY

(In seconds)

Number of failures

m	0	1	2	3
4	1.06421	1.11302	1.45934	1.73880
5	1.37995	1.39509	1.45060	1.48640
7	1.96620	2.13278	2.19353	2.18232
8	2.26595	2.28465	2.63852	2.65325
10	3.19398	3.22027	3.82371	3.83791
11	3.66591	4.20082	4.06034	4.21347
13	4.94012	5.01102	5.39482	5.63978
14	5.28370	5.89880	6.06670	6.09805
16	6.63168	7.13118	7.51444	7.60873
17	7.40714	8.02022	8.66145	8.74144
19	8.68546	9.45718	10.27761	10.96002
20	9.52298	11.11859	12.19756	13.05622

C.2. MEAN QUEUE LENGTH SUMMARY

	Number of Failures			
m	0	1	2	3
4	.00483	.00528	.01113	.01481
5	.00947	.00991	.00965	.01089
7	.01909	.02340	.02348	.02275
8	.02436	.02384	.03544	.03380
10	.04999	.04717	.06963	.06815
11	.06076	.08223	.07674	.07720
13	.10231	.10708	.11854	.12749
14	.11078	.13675	.14629	.14257
16	.15505	.17791	.19027	.19675
17	.17986	.20369	.23472	.23792
19	.22606	.26245	.30543	.33651
20	.26373	.33848	.37854	.41836

C.3. MEAN SYSTEM UTILIZATION

	Number of Failures			
m	0	1	2	3
4	.04820	.04915	.05721	.06242
5	.05890	.05970	.06067	.06069
7	.07972	.08303	.08154	.08094
8	.08849	.08925	.09713	.09270
10	.11286	.11266	.12128	.12125
11	.12104	.13323	.13101	.13023
13	.14673	.14673	.14988	.14971
14	.15060	.15810	.15848	.15578
16	.17136	.17441	.17497	.17924
17	.17796	.18123	.18957	.18653
19	.20226	.20770	.21000	.21111
20	.21263	.22342	.22481	.22740

D. SUMMARY OF SIMULATION RESULTS

```
Number of modules (nodes):                  4 (16)

Failed module(s):
      NONE!

Failed node(s) (including those in failed modules):
      NONE!

Mean message size:                          8000 bits

Line speed all links:                       19200 bits/sec

Mean message interarrival time:             20000 msecs

Total messages generated:                   8003

Messages delivered before stats:            4000

Messages used for statistics:               4000

Messages undelivered due to failure(s):     0

Messages left in network:                   3

Maximum queue length:                       3 msgs

Node with max queue:                        0

Link with max queue:                        4

Mean queue length:                          0.004832 msgs

Maximum path length:                        4 hops

Mean path length:                           2.3923 hops

Mean response time per message:             1064.2078 msecs

Mean delay/hop:                             444.8564 msecs

Mean transmission time/hop:                 416 msecs

Mean queueing time/hop:                     28.8564 msecs

Mean link busy time:                        61561.2031 msecs

Probability of link busy (rho):             0.048201

Probability msg does not queue:             0.951799
```

```
              SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                  5 (20)

Failed module(s):
      NONE!

Failed node(s) (including those in failed modules):
      NONE!

Mean message size:                          8000 bits

Line speed all links:                       19200 bits/sec

Mean message interarrival time:             20000 msecs

Total messages generated:                   8006

Messages delivered before stats:            4000

Messages used for statistics:               4000

Messages undelivered due to failure(s): 0

Messages left in network:                   6

Maximum queue length:                       4 msgs

Node with max queue:                        1

Link with max queue:                        4

Mean queue length:                          0.009471 msgs

Maximum path length:                        5 hops

Mean path length:                           2.9005 hops

Mean response time per message:             1379.9475 msecs

Mean delay/hop:                             475.7620 msecs

Mean transmission time/hop:                 416 msecs

Mean queueing time/hop:                     59.7620 msecs

Mean link busy time:                        60784.1992 msecs

Probability of link busy (rho):             0.058903

Probability msg does not queue:             0.941097
```

```
                SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                 7 (28)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
     NONE!

Mean message size:                         8000 bits

Line speed all links:                      19200 bits/sec

Mean message interarrival time:            20000 msecs

Total messages generated:                  8012

Messages delivered before stats:           4000

Messages used for statistics:              4000

Messages undelivered due to failure(s):    0

Messages left in network:                  12

Maximum queue length:                      6 msgs

Node with max queue:                       18

Link with max queue:                       3

Mean queue length:                         0.019092 msgs

Maximum path length:                       7 hops

Mean path length:                          3.8772 hops

Mean response time per message:            1966.1960 msecs

Mean delay/hop:                            507.1111 msecs

Mean transmission time/hop:                416 msecs

Mean queueing time/hop:                    91.1111 msecs

Mean link busy time:                       57764.1328 msecs

Probability of link busy (rho):            0.079719

Probability msg does not queue:            0.920281
```

```
              SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                 8 (32)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
     NONE!

Mean message size:                         8000 bits

Line speed all links:                      19200 bits/sec

Mean message interarrival time:            20000 msecs

Total messages generated:                  8015

Messages delivered before stats:           4000

Messages used for statistics:              4000

Messages undelivered due to failure(s):    0

Messages left in network:                  15

Maximum queue length:                      7 msgs

Node with max queue:                       18

Link with max queue:                       4

Mean queue length:                         0.024363 msgs

Maximum path length:                       8 hops

Mean path length:                          4.3488 hops

Mean response time per message:            2265.9526 msecs

Mean delay/hop:                            521.0583 msecs

Mean transmission time/hop:                416 msecs

Mean queueing time/hop:                    105.0583 msecs

Mean link busy time:                       56365.5938 msecs

Probability of link busy (rho):            0.088485

Probability msg does not queue:            0.911515
```

SUMMARY OF SIMULATION RESULTS

```
Number of modules (nodes):                10 (40)

Failed module(s):
      NONE!

Failed node(s) (including those in failed modules):
      NONE!

Mean message size:                        8000 bits

Line speed all links:                     19200 bits/sec

Mean message interarrival time:           20000 msecs

Total messages generated:                 8022

Messages delivered before stats:          4000

Messages used for statistics:             4000

Messages undelivered due to failure(s): 0

Messages left in network:                 22

Maximum queue length:                     8 msgs

Node with max queue:                      11

Link with max queue:                      4

Mean queue length:                        0.049986 msgs

Maximum path length:                      10 hops

Mean path length:                         5.3597 hops

Mean response time per message:           3193.9814 msecs

Mean delay/hop:                           595.9199 msecs

Mean transmission time/hop:               416 msecs

Mean queueing time/hop:                   179.9199 msecs

Mean link busy time:                      55793.5859 msecs

Probability of link busy (rho):           0.112856

Probability msg does not queue:           0.887144
```

```
            SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                  11 (44)

Failed module(s):
      NONE!

Failed node(s) (including those in failed modules):
      NONE!

Mean message size:                          8000 bits

Line speed all links:                       19200 bits/sec

Mean message interarrival time:             20000 msecs

Total messages generated:                   8029

Messages delivered before stats:            4000

Messages used for statistics:               4000

Messages undelivered due to failure(s):     0

Messages left in network:                   29

Maximum queue length:                       9 msgs

Node with max queue:                        31

Link with max queue:                        4

Mean queue length:                          0.060764 msgs

Maximum path length:                        11 hops

Mean path length:                           5.9078 hops

Mean response time per message:             3665.9067 msecs

Mean delay/hop:                             620.5249 msecs

Mean transmission time/hop:                 416 msecs

Mean queueing time/hop:                     204.5249 msecs

Mean link busy time:                        55864.1523 msecs

Probability of link busy (rho):             0.121042

Probability msg does not queue:             0.878958
```

SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):	13 (52)
Failed module(s): NONE!	
Failed node(s) (including those in failed modules): NONE!	
Mean message size:	8000 bits
Line speed all links:	19200 bits/sec
Mean message interarrival time:	20000 msecs
Total messages generated:	8051
Messages delivered before stats:	4000
Messages used for statistics:	4000
Messages undelivered due to failure(s):	0
Messages left in network:	51
Maximum queue length:	14 msgs
Node with max queue:	9
Link with max queue:	4
Mean queue length:	0.102308 msgs
Maximum path length:	13 hops
Mean path length:	6.9240 hops
Mean response time per message:	4940.1172 msecs
Mean delay/hop:	713.4773 msecs
Mean transmission time/hop:	416 msecs
Mean queueing time/hop:	297.4773 msecs
Mean link busy time:	56526.7148 msecs
Probability of link busy (rho):	0.146728
Probability msg does not queue:	0.853272

SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):	14 (56)
Failed module(s): NONE!	
Failed node(s) (including those in failed modules): NONE!	
Mean message size:	8000 bits
Line speed all links:	19200 bits/sec
Mean message interarrival time:	20000 msecs
Total messages generated:	8054
Messages delivered before stats:	4000
Messages used for statistics:	4000
Messages undelivered due to failure(s):	0
Messages left in network:	54
Maximum queue length:	10 msgs
Node with max queue:	14
Link with max queue:	4
Mean queue length:	0.110777 msgs
Maximum path length:	14 hops
Mean path length:	7.3497 hops
Mean response time per message:	5283.6992 msecs
Mean delay/hop:	718.8950 msecs
Mean transmission time/hop:	416 msecs
Mean queueing time/hop:	302.8950 msecs
Mean link busy time:	54718.0078 msecs
Probability of link busy (rho):	0.150596
Probability msg does not queue:	0.849404

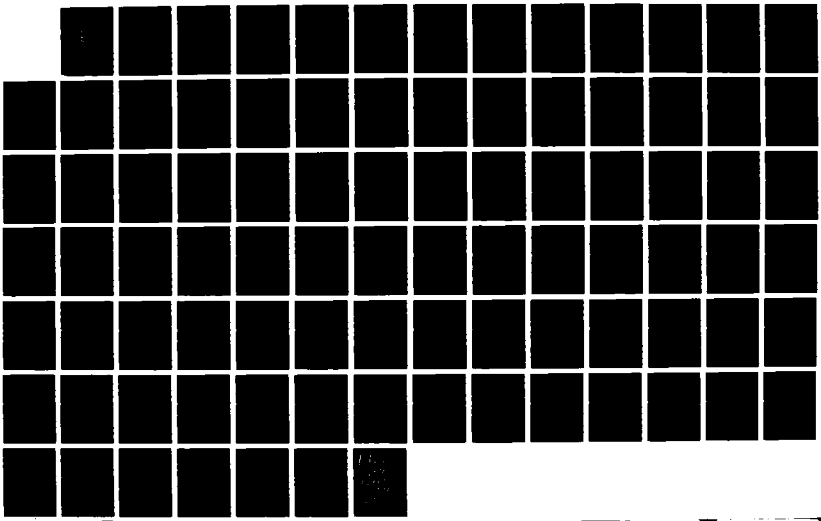
```
            SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                 16 (64)

Failed module(s):
      NONE!

Failed node(s) (including those in failed modules):
      NONE!

Mean message size:                         8000 bits

Line speed all links:                      19200 bits/sec

Mean message interarrival time:            20000 msecs

Total messages generated:                  8078

Messages delivered before stats:           4000

Messages used for statistics:              4000

Messages undelivered due to failure(s):    0

Messages left in network:                  78

Maximum queue length:                      14 msgs

Node with max queue:                       25

Link with max queue:                       4

Mean queue length:                         0.155054 msgs

Maximum path length:                       16 hops

Mean path length:                          8.4295 hops

Mean response time per message:            6631.6797 msecs

Mean delay/hop:                            786.7227 msecs

Mean transmission time/hop:                416 msecs

Mean queueing time/hop:                    370.7227 msecs

Mean link busy time:                       54123.2891 msecs

Probability of link busy (rho):            0.171358

Probability msg does not queue:            0.828642
```

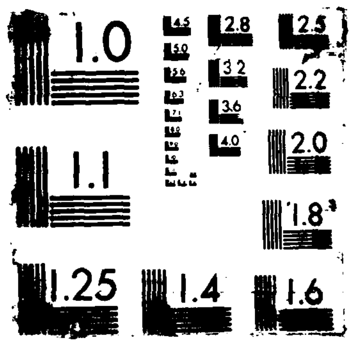
SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):	17 (68)
Failed module(s): NONE!	
Failed node(s) (including those in failed modules): NONE!	
Mean message size:	8000 bits
Line speed all links:	19200 bits/sec
Mean message interarrival time:	20000 msecs
Total messages generated:	8073
Messages delivered before stats:	4000
Messages used for statistics:	4000
Messages undelivered due to failure(s):	0
Messages left in network:	73
Maximum queue length:	19 msgs
Node with max queue:	20
Link with max queue:	4
Mean queue length:	0.179860 msgs
Maximum path length:	17 hops
Mean path length:	8.9185 hops
Mean response time per message:	7407.1406 msecs
Mean delay/hop:	830.5364 msecs
Mean transmission time/hop:	416 msecs
Mean queueing time/hop:	414.5364 msecs
Mean link busy time:	53412.2813 msecs
Probability of link busy (rho):	0.177962
Probability msg does not queue:	0.822038

AD-A196 115 A FAULT TOLERANT SELF-ROUTING COMPUTER NETWORK TOPOLOGY 3/3
(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH
T L MITCHELL 1988 AFIT/CI/NR-88-68

UNCLASSIFIED F/G 12/7 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3

```
SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                  19 (76)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
     NONE!

Mean message size:                          8000 bits

Line speed all links:                       19200 bits/sec

Mean message interarrival time:             20000 msecs

Total messages generated:                   8091

Messages delivered before stats:            4000

Messages used for statistics:               4000

Messages undelivered due to failure(s):     0

Messages left in network:                   91

Maximum queue length:                       12 msgs

Node with max queue:                        4

Link with max queue:                        4

Mean queue length:                          0.226057 msgs

Maximum path length:                        19 hops

Mean path length:                           9.8270 hops

Mean response time per message:             8685.4648 msecs

Mean delay/hop:                             883.8367 msecs

Mean transmission time/hop:                 416 msecs

Mean queueing time/hop:                     467.8367 msecs

Mean link busy time:                        54357.6641 msecs

Probability of link busy (rho):             0.202256

Probability msg does not queue:             0.797744
```

```
SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                    20 (80)

Failed module(s):
      NONE!

Failed node(s) (including those in failed modules):
      NONE!

Mean message size:                            8000 bits

Line speed all links:                         19200 bits/sec

Mean message interarrival time:               20000 msecs

Total messages generated:                     8130

Messages delivered before stats:              4000

Messages used for statistics:                 4000

Messages undelivered due to failure(s): 0

Messages left in network:                     130

Maximum queue length:                         13 msgs

Node with max queue:                          20

Link with max queue:                          4

Mean queue length:                            0.263727 msgs

Maximum path length:                          20 hops

Mean path length:                             10.2005 hops

Mean response time per message:               9522.9766 msecs

Mean delay/hop:                               933.5793 msecs

Mean transmission time/hop:                   416 msecs

Mean queueing time/hop:                       517.5793 msecs

Mean link busy time:                          53638.7500 msecs

Probability of link busy (rho):               0.212634

Probability msg does not queue:               0.787366
```

```
SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                4 (16)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
    1

Mean message size:                        8000 bits

Line speed all links:                     19200 bits/sec

Mean message interarrival time:           20000 msecs

Total messages generated:                 8541

Messages delivered before stats:          4000

Messages used for statistics:             4000

Messages undelivered due to failure(s):   539

Messages left in network:                 2

Maximum queue length:                     4 msgs

Node with max queue:                      15

Link with max queue:                      2

Mean queue length:                        0.005275 msgs

Maximum path length:                      6 hops

Mean path length:                         2.4210 hops

Mean response time per message:           1113.0168 msecs

Mean delay/hop:                           459.7344 msecs

Mean transmission time/hop:               416 msecs

Mean queueing time/hop:                   43.7344 msecs

Mean link busy time:                      70795.8750 msecs

Probability of link busy (rho):           0.049145

Probability msg does not queue:           0.950855
```

SUMMARY OF SIMULATION RESULTS

```
Number of modules (nodes):               5 (20)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
     1

Mean message size:                       8000 bits

Line speed all links:                    19200 bits/sec

Mean message interarrival time:          20000 msecs

Total messages generated:                8412

Messages delivered before stats:         4000

Messages used for statistics:            4000

Messages undelivered due to failure(s):  408

Messages left in network:                4

Maximum queue length:                    6 msgs

Node with max queue:                     3

Link with max queue:                     3

Mean queue length:                       0.009907 msgs

Maximum path length:                     6 hops

Mean path length:                        2.9467 hops

Mean response time per message:          1395.0864 msecs

Mean delay/hop:                          473.4324 msecs

Mean transmission time/hop:              416 msecs

Mean queueing time/hop:                  57.4324 msecs

Mean link busy time:                     66087.1875 msecs

Probability of link busy (rho):          0.059703

Probability msg does not queue:          0.940297
```

```
                SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                   7 (28)

Failed module(s):
      NONE!

Failed node(s) (including those in failed modules):
     1

Mean message size:                           8000 bits

Line speed all links:                        19200 bits/sec

Mean message interarrival time:              20000 msecs

Total messages generated:                    8324

Messages delivered before stats:             4000

Messages used for statistics:                4000

Messages undelivered due to failure(s): 321

Messages left in network:                    3

Maximum queue length:                        7 msgs

Node with max queue:                         18

Link with max queue:                         4

Mean queue length:                           0.023401 msgs

Maximum path length:                         8 hops

Mean path length:                            3.9890 hops

Mean response time per message:              2132.7830 msecs

Mean delay/hop:                              534.6660 msecs

Mean transmission time/hop:                  416 msecs

Mean queueing time/hop:                      118.6660 msecs

Mean link busy time:                         64512.3320 msecs

Probability of link busy (rho):              0.083031

Probability msg does not queue:              0.916969
```

```
          SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                8 (32)

Failed module(s):
      NONE!

Failed node(s) (including those in failed modules):
     1

Mean message size:                        8000 bits

Line speed all links:                     19200 bits/sec

Mean message interarrival time:           20000 msecs

Total messages generated:                 8285

Messages delivered before stats:          4000

Messages used for statistics:             4000

Messages undelivered due to failure(s):   274

Messages left in network:                 11

Maximum queue length:                     6 msgs

Node with max queue:                      3

Link with max queue:                      3

Mean queue length:                        0.023843 msgs

Maximum path length:                      10 hops

Mean path length:                         4.4765 hops

Mean response time per message:           2284.6470 msecs

Mean delay/hop:                           510.3645 msecs

Mean transmission time/hop:               416 msecs

Mean queueing time/hop:                   94.3645 msecs

Mean link busy time:                      60035.6055 msecs

Probability of link busy (rho):           0.089249

Probability msg does not queue:           0.910751
```

SUMMARY OF SIMULATION RESULTS

```
Number of modules (nodes):                 10 (40)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
     1

Mean message size:                         8000 bits

Line speed all links:                      19200 bits/sec

Mean message interarrival time:            20000 msecs

Total messages generated:                  8234

Messages delivered before stats:           4000

Messages used for statistics:              4000

Messages undelivered due to failure(s):    213

Messages left in network:                  21

Maximum queue length:                      8 msgs

Node with max queue:                       14

Link with max queue:                       4

Mean queue length:                         0.047172 msgs

Maximum path length:                       12 hops

Mean path length:                          5.4438 hops

Mean response time per message:            3220.2656 msecs

Mean delay/hop:                            591.5527 msecs

Mean transmission time/hop:                416 msecs

Mean queueing time/hop:                    175.5527 msecs

Mean link busy time:                       59691.8906 msecs

Probability of link busy (rho):            0.112657

Probability msg does not queue:            0.887343
```

```
          SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                 11 (44)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
     1

Mean message size:                         8000 bits

Line speed all links:                      19200 bits/sec

Mean message interarrival time:            20000 msecs

Total messages generated:                  8227

Messages delivered before stats:           4000

Messages used for statistics:              4000

Messages undelivered due to failure(s):    184

Messages left in network:                  43

Maximum queue length:                      11 msgs

Node with max queue:                       3

Link with max queue:                       4

Mean queue length:                         0.082228 msgs

Maximum path length:                       12 hops

Mean path length:                          6.0747 hops

Mean response time per message:            4200.8203 msecs

Mean delay/hop:                            691.5215 msecs

Mean transmission time/hop:                416 msecs

Mean queueing time/hop:                    275.5215 msecs

Mean link busy time:                       61959.9297 msecs

Probability of link busy (rho):            0.133229

Probability msg does not queue:            0.866771
```

SUMMARY OF SIMULATION RESULTS

```
Number of modules (nodes):                    13 (52)

Failed module(s):
      NONE!

Failed node(s) (including those in failed modules):
     1

Mean message size:                            8000 bits

Line speed all links:                         19200 bits/sec

Mean message interarrival time:               20000 msecs

Total messages generated:                     8216

Messages delivered before stats:              4000

Messages used for statistics:                 4000

Messages undelivered due to failure(s): 152

Messages left in network:                     64

Maximum queue length:                         13 msgs

Node with max queue:                          2

Link with max queue:                          4

Mean queue length:                            0.107082 msgs

Maximum path length:                          14 hops

Mean path length:                             6.9545 hops

Mean response time per message:               5011.0195 msecs

Mean delay/hop:                               720.5432 msecs

Mean transmission time/hop:                   416 msecs

Mean queueing time/hop:                       304.5432 msecs

Mean link busy time:                          58951.1328 msecs

Probability of link busy (rho):               0.146726

Probability msg does not queue:               0.853274
```

```
              SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                 14 (56)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
     1

Mean message size:                         8000 bits

Line speed all links:                      19200 bits/sec

Mean message interarrival time:            20000 msecs

Total messages generated:                  8211

Messages delivered before stats:           4000

Messages used for statistics:              4000

Messages undelivered due to failure(s):    152

Messages left in network:                  59

Maximum queue length:                      14 msgs

Node with max queue:                       46

Link with max queue:                       4

Mean queue length:                         0.136753 msgs

Maximum path length:                       16 hops

Mean path length:                          7.5758 hops

Mean response time per message:            5898.8047 msecs

Mean delay/hop:                            778.6428 msecs

Mean transmission time/hop:                416 msecs

Mean queueing time/hop:                    362.6428 msecs

Mean link busy time:                       58466.6875 msecs

Probability of link busy (rho):            0.158095

Probability msg does not queue:            0.841905
```

```
          SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                16 (64)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
     1

Mean message size:                        8000 bits

Line speed all links:                     19200 bits/sec

Mean message interarrival time:           20000 msecs

Total messages generated:                 8247

Messages delivered before stats:          4000

Messages used for statistics:             4000

Messages undelivered due to failure(s):   134

Messages left in network:                 113

Maximum queue length:                     16 msgs

Node with max queue:                      3

Link with max queue:                      4

Mean queue length:                        0.177909 msgs

Maximum path length:                      18 hops

Mean path length:                         8.5378 hops

Mean response time per message:           7131.1797 msecs

Mean delay/hop:                           835.2527 msecs

Mean transmission time/hop:               416 msecs

Mean queueing time/hop:                   419.2527 msecs

Mean link busy time:                      57600.7891 msecs

Probability of link busy (rho):           0.174405

Probability msg does not queue:           0.825595
```

SUMMARY OF SIMULATION RESULTS

```
Number of modules (nodes):                17 (68)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
    1

Mean message size:                        8000 bits

Line speed all links:                     19200 bits/sec

Mean message interarrival time:           20000 msecs

Total messages generated:                 8204

Messages delivered before stats:          4000

Messages used for statistics:             4000

Messages undelivered due to failure(s):   99

Messages left in network:                 105

Maximum queue length:                     19 msgs

Node with max queue:                      3

Link with max queue:                      4

Mean queue length:                        0.203690 msgs

Maximum path length:                      18 hops

Mean path length:                         8.9870 hops

Mean response time per message:           8020.2188 msecs

Mean delay/hop:                           892.4243 msecs

Mean transmission time/hop:               416 msecs

Mean queueing time/hop:                   476.4243 msecs

Mean link busy time:                      56963.7383 msecs

Probability of link busy (rho):           0.181232

Probability msg does not queue:           0.818768
```

```
         SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                   19 (76)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
    1

Mean message size:                           8000 bits

Line speed all links:                        19200 bits/sec

Mean message interarrival time:              20000 msecs

Total messages generated:                    8233

Messages delivered before stats:             4000

Messages used for statistics:                4000

Messages undelivered due to failure(s):      109

Messages left in network:                    124

Maximum queue length:                        21 msgs

Node with max queue:                         66

Link with max queue:                         4

Mean queue length:                           0.262452 msgs

Maximum path length:                         20 hops

Mean path length:                            10.0020 hops

Mean response time per message:              9457.1758 msecs

Mean delay/hop:                              945.5283 msecs

Mean transmission time/hop:                  416 msecs

Mean queueing time/hop:                      529.5283 msecs

Mean link busy time:                         55894.4102 msecs

Probability of link busy (rho):              0.207695

Probability msg does not queue:              0.792305
```

SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):	20 (80)
Failed module(s): NONE!	
Failed node(s) (including those in failed modules): 1	
Mean message size:	8000 bits
Line speed all links:	19200 bits/sec
Mean message interarrival time:	20000 msecs
Total messages generated:	8261
Messages delivered before stats:	4000
Messages used for statistics:	4000
Messages undelivered due to failure(s):	107
Messages left in network:	154
Maximum queue length:	34 msgs
Node with max queue:	70
Link with max queue:	4
Mean queue length:	0.338484 msgs
Maximum path length:	22 hops
Mean path length:	10.5735 hops
Mean response time per message:	11118.5859 msecs
Mean delay/hop:	1051.5520 msecs
Mean transmission time/hop:	416 msecs
Mean queueing time/hop:	635.5520 msecs
Mean link busy time:	56719.5430 msecs
Probability of link busy (rho):	0.223421
Probability msg does not queue:	0.776579

```
          SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                   4 (16)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
    1   6

Mean message size:                           8000 bits

Line speed all links:                        19200 bits/sec

Mean message interarrival time:              20000 msecs

Total messages generated:                    9176

Messages delivered before stats:             4000

Messages used for statistics:                4000

Messages undelivered due to failure(s):      1172

Messages left in network:                    4

Maximum queue length:                        6 msgs

Node with max queue:                         11

Link with max queue:                         4

Mean queue length:                           0.011125 msgs

Maximum path length:                         10 hops

Mean path length:                            3.0717 hops

Mean response time per message:              1459.3381 msecs

Mean delay/hop:                              475.0837 msecs

Mean transmission time/hop:                  416 msecs

Mean queueing time/hop:                      59.0837 msecs

Mean link busy time:                         96035.6875 msecs

Probability of link busy (rho):              0.057209

Probability msg does not queue:              0.942791
```

```
            SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                  5 (20)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
    1   6

Mean message size:                          8000 bits

Line speed all links:                       19200 bits/sec

Mean message interarrival time:             20000 msecs

Total messages generated:                   8930

Messages delivered before stats:            4000

Messages used for statistics:               4000

Messages undelivered due to failure(s):     926

Messages left in network:                   4

Maximum queue length:                       4 msgs

Node with max queue:                        3

Link with max queue:                        3

Mean queue length:                          0.009645 msgs

Maximum path length:                        6 hops

Mean path length:                           3.0833 hops

Mean response time per message:             1450.6011 msecs

Mean delay/hop:                             470.4780 msecs

Mean transmission time/hop:                 416 msecs

Mean queueing time/hop:                     54.4780 msecs

Mean link busy time:                        75300.8750 msecs

Probability of link busy (rho):             0.060669

Probability msg does not queue:             0.939331
```

```
          SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                 7 (28)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
    1   6

Mean message size:                         8000 bits

Line speed all links:                      19200 bits/sec

Mean message interarrival time:            20000 msecs

Total messages generated:                  8672

Messages delivered before stats:           4000

Messages used for statistics:              4000

Messages undelivered due to failure(s):    664

Messages left in network:                  8

Maximum queue length:                      6 msgs

Node with max queue:                       3

Link with max queue:                       3

Mean queue length:                         0.023479 msgs

Maximum path length:                       8 hops

Mean path length:                          4.1257 hops

Mean response time per message:            2193.5305 msecs

Mean delay/hop:                            531.6685 msecs

Mean transmission time/hop:                416 msecs

Mean queueing time/hop:                    115.6685 msecs

Mean link busy time:                       69719.1875 msecs

Probability of link busy (rho):            0.081541

Probability msg does not queue:            0.918459
```

```
        SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                8 (32)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
     1   6

Mean message size:                        8000 bits

Line speed all links:                     19200 bits/sec

Mean message interarrival time:           20000 msecs

Total messages generated:                 8569

Messages delivered before stats:          4000

Messages used for statistics:             4000

Messages undelivered due to failure(s):   558

Messages left in network:                 11

Maximum queue length:                     8 msgs

Node with max queue:                      4

Link with max queue:                      4

Mean queue length:                        0.035435 msgs

Maximum path length:                      10 hops

Mean path length:                         4.5968 hops

Mean response time per message:           2638.5215 msecs

Mean delay/hop:                           573.9971 msecs

Mean transmission time/hop:               416 msecs

Mean queueing time/hop:                   157.9971 msecs

Mean link busy time:                      68497.0625 msecs

Probability of link busy (rho):           0.097125

Probability msg does not queue:           0.902875
```

```
SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                   10 (40)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
    1   6

Mean message size:                           8000 bits

Line speed all links:                        19200 bits/sec

Mean message interarrival time:              20000 msecs

Total messages generated:                    8462

Messages delivered before stats:             4000

Messages used for statistics:                4000

Messages undelivered due to failure(s):      443

Messages left in network:                    19

Maximum queue length:                        10 msgs

Node with max queue:                         14

Link with max queue:                         4

Mean queue length:                           0.069628 msgs

Maximum path length:                         12 hops

Mean path length:                            5.7025 hops

Mean response time per message:              3823.7053 msecs

Mean delay/hop:                              670.5313 msecs

Mean transmission time/hop:                  416 msecs

Mean queueing time/hop:                      254.5313 msecs

Mean link busy time:                         66301.7500 msecs

Probability of link busy (rho):              0.121283

Probability msg does not queue:              0.878717
```

```
           SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                 11 (44)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
     1   6

Mean message size:                         8000 bits

Line speed all links:                      19200 bits/sec

Mean message interarrival time:            20000 msecs

Total messages generated:                  8399

Messages delivered before stats:           4000

Messages used for statistics:              4000

Messages undelivered due to failure(s):    361

Messages left in network:                  38

Maximum queue length:                      9 msgs

Node with max queue:                       3

Link with max queue:                       4

Mean queue length:                         0.076738 msgs

Maximum path length:                       12 hops

Mean path length:                          6.0798 hops

Mean response time per message:            4060.3425 msecs

Mean delay/hop:                            667.8469 msecs

Mean transmission time/hop:                416 msecs

Mean queueing time/hop:                    251.8469 msecs

Mean link busy time:                       63483.0313 msecs

Probability of link busy (rho):            0.131012

Probability msg does not queue:            0.868988
```

```
        SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):              13 (52)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
    1   6

Mean message size:                      8000 bits

Line speed all links:                   19200 bits/sec

Mean message interarrival time:         20000 msecs

Total messages generated:               8403

Messages delivered before stats:        4000

Messages used for statistics:           4000

Messages undelivered due to failure(s): 342

Messages left in network:               61

Maximum queue length:                   13 msgs

Node with max queue:                    28

Link with max queue:                    4

Mean queue length:                      0.118544 msgs

Maximum path length:                    14 hops

Mean path length:                       7.1685 hops

Mean response time per message:         5394.8242 msecs

Mean delay/hop:                         752.5735 msecs

Mean transmission time/hop:             416 msecs

Mean queueing time/hop:                 336.5735 msecs

Mean link busy time:                    63276.5391 msecs

Probability of link busy (rho):         0.149884

Probability msg does not queue:         0.850116
```

SUMMARY OF SIMULATION RESULTS

```
Number of modules (nodes):                   14 (56)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
    1   6

Mean message size:                           8000 bits

Line speed all links:                        19200 bits/sec

Mean message interarrival time:              20000 msecs

Total messages generated:                    8399

Messages delivered before stats:             4000

Messages used for statistics:                4000

Messages undelivered due to failure(s):      312

Messages left in network:                    87

Maximum queue length:                        14 msgs

Node with max queue:                         46

Link with max queue:                         4

Mean queue length:                           0.146285 msgs

Maximum path length:                         16 hops

Mean path length:                            7.6012 hops

Mean response time per message:              6066.6914 msecs

Mean delay/hop:                              798.1174 msecs

Mean transmission time/hop:                  416 msecs

Mean queueing time/hop:                      382.1174 msecs

Mean link busy time:                         61233.7227 msecs

Probability of link busy (rho):              0.158483

Probability msg does not queue:              0.841517
```

SUMMARY OF SIMULATION RESULTS

```
Number of modules (nodes):                      16 (64)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
    1   6

Mean message size:                              8000 bits

Line speed all links:                           19200 bits/sec

Mean message interarrival time:                 20000 msecs

Total messages generated:                       8379

Messages delivered before stats:                4000

Messages used for statistics:                   4000

Messages undelivered due to failure(s):         263

Messages left in network:                       116

Maximum queue length:                           24 msgs

Node with max queue:                            4

Link with max queue:                            4

Mean queue length:                              0.190274 msgs

Maximum path length:                            18 hops

Mean path length:                               8.5920 hops

Mean response time per message:                 7514.4414 msecs

Mean delay/hop:                                 874.5857 msecs

Mean transmission time/hop:                     416 msecs

Mean queueing time/hop:                         458.5857 msecs

Mean link busy time:                            60081.3164 msecs

Probability of link busy (rho):                 0.174970

Probability msg does not queue:                 0.825030
```

```
          SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):               17 (68)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
    1   6

Mean message size:                       8000 bits

Line speed all links:                    19200 bits/sec

Mean message interarrival time:          20000 msecs

Total messages generated:                8371

Messages delivered before stats:         4000

Messages used for statistics:            4000

Messages undelivered due to failure(s):  242

Messages left in network:                129

Maximum queue length:                    20 msgs

Node with max queue:                     3

Link with max queue:                     4

Mean queue length:                       0.234719 msgs

Maximum path length:                     18 hops

Mean path length:                        9.2028 hops

Mean response time per message:          8661.4492 msecs

Mean delay/hop:                          941.1804 msecs

Mean transmission time/hop:              416 msecs

Mean queueing time/hop:                  525.1804 msecs

Mean link busy time:                     59695.0625 msecs

Probability of link busy (rho):          0.189571

Probability msg does not queue:          0.810429
```

```
                    SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                    19 (76)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
    1   6

Mean message size:                            8000 bits

Line speed all links:                         19200 bits/sec

Mean message interarrival time:               20000 msecs

Total messages generated:                     8403

Messages delivered before stats:              4000

Messages used for statistics:                 4000

Messages undelivered due to failure(s):       222

Messages left in network:                     181

Maximum queue length:                         27 msgs

Node with max queue:                          4

Link with max queue:                          4

Mean queue length:                            0.305433 msgs

Maximum path length:                          20 hops

Mean path length:                             10.1257 hops

Mean response time per message:               10277.6055 msecs

Mean delay/hop:                               1014.9968 msecs

Mean transmission time/hop:                   416 msecs

Mean queueing time/hop:                       598.9968 msecs

Mean link busy time:                          58196.9180 msecs

Probability of link busy (rho):               0.209996

Probability msg does not queue:               0.790004
```

```
            SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                   20 (80)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
    1   6

Mean message size:                           8000 bits

Line speed all links:                        19200 bits/sec

Mean message interarrival time:              20000 msecs

Total messages generated:                    8382

Messages delivered before stats:             4000

Messages used for statistics:                4000

Messages undelivered due to failure(s):      195

Messages left in network:                    187

Maximum queue length:                        26 msgs

Node with max queue:                         70

Link with max queue:                         4

Mean queue length:                           0.378540 msgs

Maximum path length:                         22 hops

Mean path length:                            10.6435 hops

Mean response time per message:              12197.5586 msecs

Mean delay/hop:                              1146.0100 msecs

Mean transmission time/hop:                  416 msecs

Mean queueing time/hop:                      730.0100 msecs

Mean link busy time:                         58287.7930 msecs

Probability of link busy (rho):              0.224814

Probability msg does not queue:              0.775186
```

```
            SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                  4 (16)

Failed module(s):
      NONE!

Failed node(s) (including those in failed modules):
     1   6  11

Mean message size:                          8000 bits

Line speed all links:                       19200 bits/sec

Mean message interarrival time:             20000 msecs

Total messages generated:                   9907

Messages delivered before stats:            4000

Messages used for statistics:               4000

Messages undelivered due to failure(s):     1903

Messages left in network:                   4

Maximum queue length:                       7 msgs

Node with max queue:                        12

Link with max queue:                        4

Mean queue length:                          0.014812 msgs

Maximum path length:                        10 hops

Mean path length:                           3.3982 hops

Mean response time per message:             1738.7966 msecs

Mean delay/hop:                             511.6741 msecs

Mean transmission time/hop:                 416 msecs

Mean queueing time/hop:                     95.6741 msecs

Mean link busy time:                        120257.7500 msecs

Probability of link busy (rho):             0.062417

Probability msg does not queue:             0.937583
```

```
SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                5 (20)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
    1   6  11

Mean message size:                        8000 bits

Line speed all links:                     19200 bits/sec

Mean message interarrival time:           20000 msecs

Total messages generated:                 9470

Messages delivered before stats:          4000

Messages used for statistics:             4000

Messages undelivered due to failure(s):   1463

Messages left in network:                 7

Maximum queue length:                     6 msgs

Node with max queue:                      16

Link with max queue:                      4

Mean queue length:                        0.010891 msgs

Maximum path length:                      7 hops

Mean path length:                         3.1080 hops

Mean response time per message:           1486.4016 msecs

Mean delay/hop:                           478.2502 msecs

Mean transmission time/hop:               416 msecs

Mean queueing time/hop:                   62.2502 msecs

Mean link busy time:                      83398.1250 msecs

Probability of link busy (rho):           0.060693

Probability msg does not queue:           0.939307
```

```
              SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                 7 (28)

Failed module(s):
      NONE!

Failed node(s) (including those in failed modules):
     1   6  11

Mean message size:                         8000 bits

Line speed all links:                      19200 bits/sec

Mean message interarrival time:            20000 msecs

Total messages generated:                  8987

Messages delivered before stats:           4000

Messages used for statistics:              4000

Messages undelivered due to failure(s): 983

Messages left in network:                  4

Maximum queue length:                      6 msgs

Node with max queue:                       3

Link with max queue:                       4

Mean queue length:                         0.022754 msgs

Maximum path length:                       8 hops

Mean path length:                          4.2035 hops

Mean response time per message:            2182.3206 msecs

Mean delay/hop:                            519.1675 msecs

Mean transmission time/hop:                416 msecs

Mean queueing time/hop:                    103.1675 msecs

Mean link busy time:                       74489.6250 msecs

Probability of link busy (rho):            0.080938

Probability msg does not queue:            0.919062
```

```
            SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                   8 (32)

Failed module(s):
      NONE!

Failed node(s) (including those in failed modules):
     1   6  11

Mean message size:                           8000 bits

Line speed all links:                        19200 bits/sec

Mean message interarrival time:              20000 msecs

Total messages generated:                    8868

Messages delivered before stats:             4000

Messages used for statistics:                4000

Messages undelivered due to failure(s):      856

Messages left in network:                    12

Maximum queue length:                        7 msgs

Node with max queue:                         23

Link with max queue:                         4

Mean queue length:                           0.033797 msgs

Maximum path length:                         10 hops

Mean path length:                            4.6625 hops

Mean response time per message:             2653.2473 msecs

Mean delay/hop:                              569.0610 msecs

Mean transmission time/hop:                  416 msecs

Mean queueing time/hop:                      153.0610 msecs

Mean link busy time:                         72599.4375 msecs

Probability of link busy (rho):              0.092701

Probability msg does not queue:              0.907299
```

```
            SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                  10 (40)

Failed module(s):
      NONE!

Failed node(s) (including those in failed modules):
     1   6  11

Mean message size:                          8000 bits

Line speed all links:                       19200 bits/sec

Mean message interarrival time:             20000 msecs

Total messages generated:                   8690

Messages delivered before stats:            4000

Messages used for statistics:               4000

Messages undelivered due to failure(s): 667

Messages left in network:                   23

Maximum queue length:                       10 msgs

Node with max queue:                        37

Link with max queue:                        4

Mean queue length:                          0.068149 msgs

Maximum path length:                        12 hops

Mean path length:                           5.7530 hops

Mean response time per message:             3837.9099 msecs

Mean delay/hop:                             667.1145 msecs

Mean transmission time/hop:                 416 msecs

Mean queueing time/hop:                     251.1145 msecs

Mean link busy time:                        70789.5000 msecs

Probability of link busy (rho):             0.121248

Probability msg does not queue:             0.878752
```

```
SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                   11 (44)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
     1   6  11

Mean message size:                           8000 bits

Line speed all links:                        19200 bits/sec

Mean message interarrival time:              20000 msecs

Total messages generated:                    8611

Messages delivered before stats:             4000

Messages used for statistics:                4000

Messages undelivered due to failure(s): 570

Messages left in network:                    41

Maximum queue length:                        10 msgs

Node with max queue:                         16

Link with max queue:                         4

Mean queue length:                           0.077202 msgs

Maximum path length:                         13 hops

Mean path length:                            6.3127 hops

Mean response time per message:              4213.4688 msecs

Mean delay/hop:                              667.4536 msecs

Mean transmission time/hop:                  416 msecs

Mean queueing time/hop:                      251.4536 msecs

Mean link busy time:                         68500.6250 msecs

Probability of link busy (rho):              0.130229

Probability msg does not queue:              0.869771
```

```
                SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                 13 (52)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
    1   6  11

Mean message size:                         8000 bits

Line speed all links:                      19200 bits/sec

Mean message interarrival time:            20000 msecs

Total messages generated:                  8519

Messages delivered before stats:           4000

Messages used for statistics:              4000

Messages undelivered due to failure(s):    461

Messages left in network:                  58

Maximum queue length:                      16 msgs

Node with max queue:                       5

Link with max queue:                       4

Mean queue length:                         0.127487 msgs

Maximum path length:                       15 hops

Mean path length:                          7.2198 hops

Mean response time per message:            5639.7773 msecs

Mean delay/hop:                            781.1594 msecs

Mean transmission time/hop:                416 msecs

Mean queueing time/hop:                    365.1594 msecs

Mean link busy time:                       65324.0664 msecs

Probability of link busy (rho):            0.149708

Probability msg does not queue:            0.850292
```

```
        SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                14 (56)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
     1   6  11

Mean message size:                        8000 bits

Line speed all links:                     19200 bits/sec

Mean message interarrival time:           20000 msecs

Total messages generated:                 8531

Messages delivered before stats:          4000

Messages used for statistics:             4000

Messages undelivered due to failure(s):   454

Messages left in network:                 77

Maximum queue length:                     17 msgs

Node with max queue:                      52

Link with max queue:                      4

Mean queue length:                        0.142565 msgs

Maximum path length:                      16 hops

Mean path length:                         7.6517 hops

Mean response time per message:           6098.0508 msecs

Mean delay/hop:                           796.9485 msecs

Mean transmission time/hop:               416 msecs

Mean queueing time/hop:                   380.9485 msecs

Mean link busy time:                      62550.7070 msecs

Probability of link busy (rho):           0.155780

Probability msg does not queue:           0.844220
```

SUMMARY OF SIMULATION RESULTS

```
Number of modules (nodes):                 16 (64)

Failed module(s):
      NONE!

Failed node(s) (including those in failed modules):
     1   6  11

Mean message size:                         8000 bits

Line speed all links:                      19200 bits/sec

Mean message interarrival time:            20000 msecs

Total messages generated:                  8478

Messages delivered before stats:           4000

Messages used for statistics:              4000

Messages undelivered due to failure(s):    386

Messages left in network:                  92

Maximum queue length:                      20 msgs

Node with max queue:                       3

Link with max queue:                       4

Mean queue length:                         0.196746 msgs

Maximum path length:                       18 hops

Mean path length:                          8.6327 hops

Mean response time per message:            7608.7266 msecs

Mean delay/hop:                            881.3792 msecs

Mean transmission time/hop:                416 msecs

Mean queueing time/hop:                    465.3792 msecs

Mean link busy time:                       61365.2461 msecs

Probability of link busy (rho):            0.179242

Probability msg does not queue:            0.820758
```

```
SUMMARY OF SIMULATION RESULTS

Number of modules (nodes):                  17 (68)

Failed module(s):
     NONE!

Failed node(s) (including those in failed modules):
     1   6  11

Mean message size:                          8000 bits

Line speed all links:                       19200 bits/sec

Mean message interarrival time:             20000 msecs

Total messages generated:                   8482

Messages delivered before stats:            4000

Messages used for statistics:               4000

Messages undelivered due to failure(s): 394

Messages left in network:                   88

Maximum queue length:                       19 msgs

Node with max queue:                        17

Link with max queue:                        4

Mean queue length:                          0.237917 msgs

Maximum path length:                        19 hops

Mean path length:                           9.1823 hops

Mean response time per message:             8741.4414 msecs

Mean delay/hop:                             951.9932 msecs

Mean transmission time/hop:                 416 msecs

Mean queueing time/hop:                     535.9932 msecs

Mean link busy time:                        61513.1406 msecs

Probability of link busy (rho):             0.186532

Probability msg does not queue:             0.813468
```

SUMMARY OF SIMULATION RESULTS

```
Number of modules (nodes):                  19 (76)

Failed module(s):
      NONE!

Failed node(s) (including those in failed modules):
     1   6  11

Mean message size:                          8000 bits

Line speed all links:                       19200 bits/sec

Mean message interarrival time:             20000 msecs

Total messages generated:                   8524

Messages delivered before stats:            4000

Messages used for statistics:               4000

Messages undelivered due to failure(s):     328

Messages left in network:                   196

Maximum queue length:                       26 msgs

Node with max queue:                        3

Link with max queue:                        4

Mean queue length:                          0.336510 msgs

Maximum path length:                        21 hops

Mean path length:                           10.2530 hops

Mean response time per message:             10960.0195 msecs

Mean delay/hop:                             1068.9573 msecs

Mean transmission time/hop:                 416 msecs

Mean queueing time/hop:                     652.9573 msecs

Mean link busy time:                        60174.3555 msecs

Probability of link busy (rho):             0.211112

Probability msg does not queue:             0.788888
```

SUMMARY OF SIMULATION RESULTS

```
Number of modules (nodes):                 20 (80)

Failed module(s):
      NONE!

Failed node(s) (including those in failed modules):
     1   6  11

Mean message size:                         8000 bits

Line speed all links:                      19200 bits/sec

Mean message interarrival time:            20000 msecs

Total messages generated:                  8481

Messages delivered before stats:           4000

Messages used for statistics:              4000

Messages undelivered due to failure(s):    319

Messages left in network:                  162

Maximum queue length:                      27 msgs

Node with max queue:                       70

Link with max queue:                       4

Mean queue length:                         0.418360 msgs

Maximum path length:                       22 hops

Mean path length:                          10.7830 hops

Mean response time per message:            13056.2188 msecs

Mean delay/hop:                            1210.8149 msecs

Mean transmission time/hop:                416 msecs

Mean queueing time/hop:                    794.8149 msecs

Mean link busy time:                       60823.9492 msecs

Probability of link busy (rho):            0.227400

Probability msg does not queue:            0.772600
```

```
                /* E.  SIMULATION MODEL SOURCE CODE LISTING  */
#include <stdio.h>
#include <math.h>
#define YES     'y'
#define CR       012
#define ENUFD    0444                  /* to stop adding node numbers */
#define ENUF     444                   /* to stop inputting failures  */
#define NO      'n'
#define LOWER_NODE    5                /* to find start nd link col   */
#define iSTEP         4                /* each module has four nodes  */
#define kSTEP         3                /* to connect node in resp col */
#define LOWER_PLACE   7                /* to access position columns  */
#define MAX_COLUMNS   19               /* keeps track of matrix col   */
#define LINK_ND_COL   16               /* finds next nd col for links */
#define NXT_MDL_COL   14               /* finds next mdl col for links*/
#define NXT_ND_POSN   18               /* finds next node pos'n col   */
#define POSITION_CALC (j + 13) % nodes /* used to find link nd pos'n  */
#define LARGEST       1024             /* max size of msg_area array  */
#define INFINITY      1999999999             /* very large number     */
#define SERVICE_TIME  (length * 1000)/RATE   /* time is in millisecs  */

FILE *outfile;                         /* save current n/wk topology  */
FILE *fopen();
FILE *infile;                          /* restore cur n/wk topology   */
FILE *snapout;                         /* snapshot of cur topology    */
FILE *savefile;                        /* write to stats file         */
FILE *statsfile;                       /* write to summary at end     */
FILE *frequency;                       /* write to destination file   */
FILE *graphit;                         /* write to graphs file        */

     int POSITION[256][19];
                                       /* used to build network       */
                                       /* 0 - position number         */
                                       /* 1 - home node number        */
                                       /* 2 - status of that node     */
                                       /* 3 - home module number      */
                                       /* 4 - status of that module   */
                                       /* 5 - directly connected      */
                                       /* local node number           */
                                       /* 6 - status that node        */
                                       /* 7 - pos'n number that node  */
                                       /* 8 - next dir con local nd   */
                                       /* 9 - status that node        */
                                       /* 10 - pos'n nmbr that node   */
                                       /* 11 - last dir con local nd  */
                                       /* 12 - status that node       */
                                       /* 13 - pos'n nmbr that node   */
                                       /* 14 - link module number     */
                                       /* 15 - status link module     */
                                       /* 16 - link node that module  */
                                       /* 17 - status of link node    */
                                       /* 18 - pos'n number link nd   */
```

```
int PTR_ARRAY[6][2];                  /* 3 rows to chg top/bot ptrs  */

int nodes;                            /* no. of nodes in network     */
int max_modules;                      /* max number of mdl desired   */
int max_numbers;                      /* number of node #'s avail    */
int cur_modules;                      /* number of mdls in the n/wk  */
int start;                            /* starting # to expand n/wk   */

int FREQ[256][2];                     /* destination address count   */
                                      /* 0 - node #, 1 - msgs rx     */
                                      /* updated by save_stats func  */
int GFEL[64][3];                      /* global future events list   */
                                      /* 0 - module  number          */
                                      /* 1 - min time of each module */
                                      /* 2 - node with min time      */
int MSG_AREA[LARGEST][40];            /* message work area           */
                                      /* 0 - position number         */
                                      /* 1 - row availability flag   */
                                      /* 2 - message number          */
                                      /* 3 - message size            */
                                      /* 4 - spiral flag this msg    */
                                      /* 5 - direction flag          */
                                      /* 6 - destination node        */
                                      /* 7 - destination module      */
                                      /* 8 - send time               */
                                      /* 9 - receive time            */
                                      /* 10 - mdl   # of spiral chg  */
                                      /* 11 - mdl # of direc'n chg   */
                                      /* 12 - route trace pointer    */
                                      /* 13-39 = route trace area    */
int NODE_AREA[LARGEST][60];           /* node level svr & q area     */
                                      /* 0 - row number              */
                                      /* 1 - node number             */
                                      /* 2 - server number (1-4)     */
                                      /* 3 - server status (0 or 1)  */
                                      /* 4 - arrival time for FEL    */
                                      /* 5 - depart time for FEL     */
                                      /* 6 - minimum of 4 and 5      */
                                      /* 7 - server queue pointer    */
                                      /* 8 - MSG_AREA loc this msg   */
                                      /* 9 - 59 = available Q slots  */
int DIR[64][2];                       /* used to set DF,             */
int BEGIN;                            /* used to set DF, source      */
int END;                              /* used to set DF, destination */
float STATS[LARGEST][5];              /* contains simulation results */
                                      /*  0 - node_area row number   */
                                      /*  1 - server max-q length    */
                                      /*  2 - svr total busy time    */
                                      /*  3 - used for msg size graf */
                                      /*  4 - available for use      */

int imevent;                          /* contains the imminent event */
int min_row;                          /* row where imevent is        */
```

```
     int min_time;                     /* time of imminent event      */
     int CLOCK;
     int RATE;                         /* contains the line speed     */
     int mean_size;
     int IAT;                          /* mean msg interarrival time  */
     int STABILIZE;                    /* used to get past startup    */
     int no_done;                      /* no msgs delivered so far    */

     int no_killed;                    /* contains # undeliv'd msgs   */
     int max_msgs;                     /* stopping criteria           */
     int msg_no;                       /* number of individual msgs   */
     int location;                     /* location in msg work array  */
     int max_path;                     /* has the largest path number */
     int stop_it;                      /* stop run if no more q-slots */
                                       /* if events not at save time  */
                                       /* used in set_params function */
     int dir1;                         /* used to setup DF mtx 1 time */
     int clk_first_msg;                /* time first msg is delivered */
     int clk_last_msg;                 /* time last msg is delivered  */
     float total_hops;                 /* used for ave. path length   */
     int resp_time;                    /* used to find ave. n/w delay */
     float ave_q_length;               /* running total of all q_lens */

main()

{

     int c;
     int i;
     int k;

     for (i = 0; i < 50; ++i)
          printf("\n");

printf("\t**********************************************************\n");
printf("\t*                                                        *\n");
printf("\t* WELCOME TO THE SIMULATION OF A NEW, EASILY EXPANDABLE *\n");
printf("\t* FAULT TOLERANT GENERAL PURPOSE SELF-ROUTING COMPUTER  *\n");
printf("\t*  COMMUNICATIONS NETWORK TOPOLOGY!  WE BITS DO HOPE    *\n");
printf("\t*     YOUR EXPERIENCE WITH US IS A PLEASANT ONE.        *\n");
printf("\t*                                                        *\n");
printf("\t**********************************************************\n");

     printf("\n\n\n\n\n\n\n");
     printf("\t\t\tPRESS RETURN TO CONTINUE!     ");
     c = getchar();

     k = 0;
     restore();
     if (cur_modules == 4)
          set_pointers();
     clear_failures();
     printf("\n\nThere are no failed nodes or modules in the network.");
     printf("\n\nAll previous failures have been cleared!\n");
     current_status();
     snapshot();
     printf("\nThe current largest node number ");
     printf("is %o", POSITION[max_numbers-1][1]);
     printf(" at position number %d.\n", max_numbers-1);

repeat1: printf("\nDo you want to add node numbers?     ");
ignore1: c = getchar();
      if (c == YES)         {
          k = node_nos();

repeat5:      printf("\nDid you enter the number(s) correctly?      ");
ignore5:      c = getchar();
          if (c == NO)
               goto repeat1;

          else if (c == CR)
               goto ignore5;

          else if (c != YES)     {
               printf("\nThat was not a valid response!\n");
               goto repeat5;
                              }
                         }

     else if (c == CR)
          goto ignore1;

     else if (c != NO)                 {
          printf("\nThat was not a valid response.\n");
          goto repeat1;
                                       }

      if (k != 0)
           max_numbers = k;

      if (cur_modules != 4)          {
repeat3: printf("\nDo you want to set up the ");
        printf("minimum 4 module network?     ");
ignore3: c = getchar();

     if (c == YES)     {
          start = 0;
          max_modules = 4;
          cur_modules = 0;
          for (i = 0; i < 4; ++i)
               nodes = build();               /* build four indep mdls */
          next_module();                      /* connect them together */
          set_pointers();                     /* set for more expans'n */
          current_status();
          snapshot();
                      }
     else if (c == CR)
          goto ignore3;

     else if (c != NO)     {
          printf("\nThat was not a valid response!\n");
          goto repeat3;
                                  }
                                            }

repeat7: printf("\nDo you want to alter ");
         printf("the number of modules(nodes)?     ");
ignore7:      c = getchar();
          if (c == YES)
               alter();

          else if (c == CR)
               goto ignore7;

          else if (c != NO)
                                  {
          printf("\nThat was not a valid response!\n");
          goto repeat7;
                                  }

/* The following portion of the program sets up the existing topology */
/* with failures if desired, and prepares to send messages over the   */
/* network.                                                            */

repeat10:  printf("\nDo you want the operational network to have");
           printf(" failed nodes or modules?   ");
ignore10:  c = getchar();

       if (c == YES)     {
          failure_setup();
          current_status();
          snapshot();
                           }

       else if (c == CR)
          goto ignore10;

       else if (c != NO)     {
          printf("\nThat was not a valid response!\n");
          goto repeat10;
                                }

repeat8: printf("\nAre you ready to send ");
         printf("traffic over the network?    ");
ignore8: c = getchar();

       if (c == NO)               {
repeat9:     printf("\nDo you want to alter the network?    ");
```

```
ignore9:     c = getchar();
               if (c == YES)     {
                    clear_failures();
                    printf("\nAll previous failures ");
                    printf("have been cleared!\n");
                    alter();
                    goto repeat10;
                                   }

               else if (c  == CR)
                    goto ignore9;

               else if (c == NO)
                    goto end;

               else           {
                    printf("\nThat was not a valid response!");
                    goto repeat9;
                              }
                                  }

     else if (c == CR)
          goto ignore8;

     else if (c != YES)     {
          printf("\nThat was not a valid response!\n");
          goto repeat8;
                            }

     simulate();
end:     save();

}


node_nos()

/* This function is used to load node numbers into the matrix.  It is */
/* called by the main network control program.                        */

{
     int i;
     int data;

     i = max_numbers;
     scanf("\n%o", &data);
     while(data != ENUFO)     {
          POSITION[i][1] = data;
          printf("\n\t%6d %6o     ", i, POSITION[i][1]);
          ++i;
          scanf("%o", &data);
                              }
     return(i);
```

```
}


build()

/* This function is used to build the fully connected modules of 4    */
/* nds each.  A separate function (next_module or add_module) adds    */
/* these modules to the network.  It's called by the main network ctl */
/* program, the alter, and partition functions.                       */

{
     int i;
     int j;
     int temp;
     int k;
     int l;
     int upper;
     int location;
     int home_module;

     i = start;
     upper = i + iSTEP;

     while (i < upper)     {
          k = LOWER_NODE;
          j = 0;
          l = LOWER_PLACE;
          temp = POSITION[start][l];
          location = start;
          home_module = start/4 + 1;

          while (j < kSTEP)     {
               if (temp != POSITION[i][l])     {
                    POSITION[i][k] = temp;
                    POSITION[i][l] = location;
                    ++j;
                    ++temp;
                    ++location;
                    k = k + kSTEP;
                    l = l + kSTEP;
                                               }
               else                 {
                    ++temp;
                    ++location;
                                    }
                                }
          POSITION[i][3] = home_module;
          ++i;
                         }
     ++cur_modules;
     start = i;
     return(i);
}
```

```
current_status()

/* This function is used to make the decision as to whether or not to */
/* print the current connectivity matrix.  It is called by the main    */
/* network control program and the alter function.                     */

{
          int c;

       printf("\nThe current number of modules ");
       printf("in the network is %d.", cur_modules);
       printf("\n\nThe current number of nodes is %d.", nodes);
       printf("\n\nThe total number of node numbers");
       printf(" available is %d.", max_numbers - nodes);
repeat4:  printf("\n\nDo you want to see the current connectivity");
       printf(" matrix?   ");

ignore4:  c = getchar();
         if (c == YES)
               print_matrix();

         else if (c == CR)
              goto ignore4;

         else if (c != NO)     {
              printf("\nThat was not a valid response!");
              goto repeat4;
                                 }
}


next_module()

/* This function connects local nds to the appropriate link nodes in  */
/* the next module for the minimum 4 module network.  Called by the   */
/* main control program.                                               */

{

     int a;
     int j;

     j = 0;
     while (j < nodes)     {

          a = POSITION_CALC;
          POSITION[j][NXT_ND_POSN] = a;
          POSITION[j][LINK_ND_COL] = POSITION[a][1];
          POSITION[j][NXT_MDL_COL] = POSITION[a][3];
          POSITION[a][NXT_ND_POSN] = j;
          POSITION[a][LINK_ND_COL] = POSITION[j][1];
          POSITION[a][NXT_MDL_COL] = POSITION[j][3];
```

```
        j = j + 2;
                             }
}


alter()

/* This function is used to decide which type of alterations will be  */
/* made to the existing network. The appropriate subfunction's called */
/* according to the decision made. Alter's called by the main network */
/* control program.                                                    */

{

     int c;
     int i;
     int j;

repeat2: printf("\nHow many modules do you want the new network");
     printf(" to have?     ");
     scanf("%d", &max_modules);
         printf("\nDid you enter the number correctly?     ");
ignore2:   c = getchar();
     if (c == NO)
          goto repeat2;

     else if (c == CR)
          goto ignore2;

     else if (c != YES)     {
          printf("\nThat was not a valid response!\n");
          goto repeat2;
                              }

     if (max_modules < 4)     {                /* invalid request      */
          printf("\nThe network must have ");
          printf("atleast 4 modules to exist!\n");
          goto repeat2;
                               }
     else if ((c = max_modules % 3) == 0)     { /* net's  partitioned */
          j = partition();
          if (j == 1)
               goto repeat2;
          else         {
               current_status();
               snapshot();
                        }
                                              }

     else if (max_modules < cur_modules)     {  /* want less modules  */
          cur_modules = 0;
          start = 0;
          for (i = 0; i < 4; ++i)
               nodes = build();
          next_module();
          set_pointers();
          while (i < max_modules)     {
               nodes = build();
               add_module();
               ++i;
                                      }
          cur_modules = i;
          current_status();
          snapshot();
                                             }

     else if (cur_modules == max_modules)     {
          printf("\nThat is how many modules you already have!\n");
          goto repeat2;
                                              }

     else if (max_modules > (max_numbers/4))     {   /* out of bounds */
          printf("\nThere are currently a total of %d", max_numbers);
          printf(" node numbers available. ");
          printf("\nCan't have %d", max_modules);
          printf("  modules.\n");
          goto repeat2;
                                                 }

     else           {
          for (i = (cur_modules +1); i <= max_modules; ++i)     {
               nodes = build();
               add_module();
                                                                }
          current_status();
          snapshot();
                    }
}


partition()

/* This function takes care of the case where the nbr of modules in    */
/* the network is a multiple of three (3). When this happens, the      */
/* network is partitioned into three separate but equal subnets. When */
/* the nbr of modules is equal to or greater than 12 and an integer    */
/* multiple of three, network performance is the same as a one 1/3     */
/* that size. Partition is called by the alter function.              */

{

        int c;
        int j;
```

```
repeat_c: printf("\n\nThe number of modules ");
        printf("you requested would leave the network");
        printf("\npartitioned  into three separate  but equal ");
        printf(" subnetworks of \n%d", max_modules/3);
        printf(" modules each.  Is that what you want?\t\t");
ignore_c:  c = getchar();
        if (c == NO)
             j = 1;

        else if (c == CR)
             goto ignore_c;

        else if (c != YES)     {
             printf("\nThat was not a valid response!\n");
             goto repeat_c;
                              }

        else      {
          printf("\n\nIf you want 6 or 9 modules, then the resulting ");
          printf("topology of 2 or \n3 modules each ");
          printf("per subnet is less than a fully connected mesh");
          printf("\ntype topology.  So ");
          printf("that is not a valid request!  If you want to ");
          printf("\nevaluate 12 or more modules whose number is an ");
          printf("integer multiple \nof 3, then performance of that");
          printf(" partitioned  network is the same");
          printf("\nas a network with 1/3 the number ");
          printf("of modules you requested.  For \nexample, a 12 ");
          printf("module network performs like the minimum 4 module");
          printf("\none, and is equal to 3 independent 4 module");
          printf(" networks.   Enter a \nrequest for the more simple");
          printf(" network.  Your entry should  be 1/3 \nof your");
          printf(" earlier request.\n\n");
          j = 1;
                 }
        return(j);

}


add_module()

/* This module adds newly generated modules to the existing network    */
/* when the request is greater than the minimum four module topology. */
/* It is called by the alter function.                                 */

{

     int temp;
     int temp1;
     int temp2;
     int temp3;
     int i;
```

```
/* The following code changes pointers & links at the top of the nwk. */

     temp1 = PTR_ARRAY[0][0];
     PTR_ARRAY[0][0] = PTR_ARRAY[1][0];
     PTR_ARRAY[1][0] = PTR_ARRAY[2][0];
     temp = start - 4;
     PTR_ARRAY[2][0] = temp;

     for (i = 0; i < 3; ++i)          {
          temp2 = PTR_ARRAY[i][0];
          temp3 = PTR_ARRAY[i][1];

          POSITION[temp2][NXT_ND_POSN] = PTR_ARRAY[i][1];
          POSITION[temp2][LINK_ND_COL] = POSITION[temp3][1];
          POSITION[temp2][NXT_MDL_COL] = POSITION[temp3][3];

          POSITION[temp3][NXT_ND_POSN] = PTR_ARRAY[i][0];
          POSITION[temp3][LINK_ND_COL] = POSITION[temp2][1];
          POSITION[temp3][NXT_MDL_COL] = POSITION[temp2][3];
                                         }

     ++temp;
     POSITION[temp1][NXT_ND_POSN] = temp;
     POSITION[temp1][LINK_ND_COL] = POSITION[temp][1];
     POSITION[temp1][NXT_MDL_COL] = POSITION[temp][3];

     POSITION[temp][NXT_ND_POSN] = temp1;
     POSITION[temp][LINK_ND_COL] = POSITION[temp1][1];
     POSITION[temp][NXT_MDL_COL] = POSITION[temp1][3];

/* The next code changes the pointers and links on the bottom of nwk  */

     temp1 = PTR_ARRAY[3][0];
     PTR_ARRAY[3][0] = PTR_ARRAY[4][0];
     PTR_ARRAY[4][0] = PTR_ARRAY[5][0];
     ++temp;
     PTR_ARRAY[5][0] = temp;

     for (i = 3; i < 6; ++i)          {
          temp2 = PTR_ARRAY[i][0];
          temp3 = PTR_ARRAY[i][1];

          POSITION[temp2][NXT_ND_POSN] = PTR_ARRAY[i][1];
          POSITION[temp2][LINK_ND_COL] = POSITION[temp3][1];
          POSITION[temp2][NXT_MDL_COL] = POSITION[temp3][3];

          POSITION[temp3][NXT_ND_POSN] = PTR_ARRAY[i][0];
          POSITION[temp3][LINK_ND_COL] = POSITION[temp2][1];
          POSITION[temp3][NXT_MDL_COL] = POSITION[temp2][3];
                                    }

     ++temp;
```

```
          POSITION[temp1][NXT_ND_POSN] = temp;
          POSITION[temp1][LINK_ND_COL] = POSITION[temp][1];
          POSITION[temp1][NXT_MDL_COL] = POSITION[temp][3];

          POSITION[temp][NXT_ND_POSN] = temp1;
          POSITION[temp][LINK_ND_COL] = POSITION[temp1][1];
          POSITION[temp][NXT_MDL_COL] = POSITION[temp1][3];
     }


     set_pointers()

     /* This function initializes pointers used to expand the minimum four */
     /* module network.  It is called by the main network control program. */

     {
          PTR_ARRAY[0][0] = 4;
          PTR_ARRAY[0][1] = 1;
          PTR_ARRAY[1][0] = 8;
          PTR_ARRAY[1][1] = 5;
          PTR_ARRAY[2][0] = 12;
          PTR_ARRAY[2][1] = 9;
          PTR_ARRAY[3][0] = 6;
          PTR_ARRAY[3][1] = 3;
          PTR_ARRAY[4][0] = 10;
          PTR_ARRAY[4][1] = 7;
          PTR_ARRAY[5][0] = 14;
          PTR_ARRAY[5][1] = 11;
     }


     failure_setup()

     /* This function sets up the network to simulate failed mdls and nds  */
     /* for network analysis.  It is called by the main network ctrl prgm. */

     {
          int c;
          int fail;
          int i;
          int j;

          int MDL_FAILURE[50][1];
          int ND_FAILURE[100][1];

     repeat11: printf("\nDo you want complete modules to fail?      ");
     ignore11: c = getchar();

               if (c == YES)     {

               printf("\nEnter modules you want to fail, ");
               printf("one module number per line.\n\n");
          i = 1;
          scanf("\n%d", &fail);
          while (fail != ENUF)     {
               MDL_FAILURE[i][0] = fail;
               printf("\n\t%6d %6d     ", i, MDL_FAILURE[i][0]);
               ++i;
               scanf("\n%d", &fail);
                                   }

repeat12:      printf("\nDid you enter the numbers correctly?     ");
ignore12:      c = getchar();

               if (c == NO)
                    goto repeat11;

               else if (c == CR)
                    goto ignore12;

               else if (c != YES)     {
                    printf("\nThat was not a valid response!\n");
                    goto repeat12;
                                      }
          j = 1;
          while (j < i)    {
               fail = MDL_FAILURE[j][0];
               if (fail < 1 || fail > cur_modules)     {
                    printf("\n\nThere is no module in the");
                    printf(" current topology numbered %4d!", fail);
                    printf("\n\nYour request is ignored");
                    printf(" since the smallest module in the");
                    printf("\n\nnetwork is one, and the largest");
                    printf(" is %4d.\n", cur_modules);
                    ++j;
                                                       }
               else           {
                    mark_module(fail);
                    ++j;
                              }
                          }

repeat14: printf("\nDo you also want nodes to fail?    ");
ignore14: c = getchar();
          if (c == YES)
               goto nodes2;
          else if (c == CR)
               goto ignore14;

          else if (c != NO)     {
               printf("\nThat was not a valid response!\n");
               goto repeat14;
                                }
                         }
```

```
     else if (c == CR)
          goto ignore11;

     else if (c != NO)     {
          printf("\nThat was not a valid response!\n");
          goto repeat11;
                              }

     else          {
nodes2:   printf("\nEnter nodes you want to fail, one");
          printf(" node number per line.\n\n ");
          i = 1;
          scanf("\n%d", &fail);
          while (fail != ENUF)     {
               ND_FAILURE[i][0] = fail;
               printf("\n\t%6d %6d     ", i, ND_FAILURE[i][0]);
               ++i;
               scanf("\n%d", &fail);
                                        }

repeat13:      printf("\nDid you enter the numbers correctly?    ");
ignore13:      c = getchar();

               if (c == NO)
                    goto repeat14;

               else if (c == CR)
                    goto ignore13;

               else if (c != YES)     {
                    printf("\nThat was not a valid response!\n");
                    goto repeat13;
                                        }

          for (j = 1; j < i; ++j)     {
               fail = ND_FAILURE[j][0];
               if (fail > nodes)          {
                   printf("\n\nThere is no node in the current");
                   printf(" topology numbered %4d!", fail);
                   printf("\n\nYour request is ignored since the ");
                   printf("largest node in the network is %4d!", nodes);
                                                }
               else
                    mark_node(fail);
                                        }
                    }
}


mark_module(x)

/* This function sets 1's in the appropriate columns of the network   */
/* connectivity matrix to show that the desired module has failed. It */
/* also calls set_links to update the contained nodes in the module.  */
/* It is called by the failure_setup function.                        */

     int x;
{
     int i;
     int j;

     i = 0;
     while (i < start)     {
          if (POSITION[i][3] == x)     {
               POSITION[i][2] = 1;
               POSITION[i][4] = 1;
                                       }
          if (POSITION[i][14] == x)
               POSITION[i][15] = 1;
          ++i;
                           }

     j = 0;
     while (j < start)     {
          if (POSITION[j][2] == 1)
               set_link(j);
          ++j;
                           }
}


mark_node(y)

/* This function sets 1's in the appropriate columns of the network   */
/* connectivity matrix to show that the desired nodes have failed. It */
/* is called by the failure_setup function.                           */

     int y;
{
     int i;

     for (i = 0; i < start; ++i)
          if (POSITION[i][1] == POSITION[y][1])     {
               POSITION[i][2] = 1;
               set_link(i);
                                                    }
}


set_link(kill)

/* This funct'n sets the appropriate links in the next module section */
/* to reflect that the desired nodes have failed. This information is */
/* needed for routing. It's called by mark_module and mark_node.      */

     int kill;
```

```
{
     int k;

     for (k = 0; k < start; ++k)      {
          if (POSITION[k][5] == POSITION[kill][1])
               POSITION[k][6] = 1;
          if (POSITION[k][8] == POSITION[kill][1])
               POSITION[k][9] = 1;
          if (POSITION[k][11] == POSITION[kill][1])
               POSITION[k][12] = 1;
          if (POSITION[k][16] == POSITION[kill][1])
               POSITION[k][17] = 1;
                                      }
}


print_matrix()

/* This function prints the network connectivity matrix which defines */
/* the complete network topology.  Called by current_status function. */

{

     int i;
     int j;

     printf("\n\n\n\n\n\n\n\n\n\n\n\t                    ");
     printf("NETWORK CONNECTIVITY ");
     printf(" MATRIX\n\n\n\n\n\n\n");
     printf("\n\n POS'N    HOME     HOME                 ");
     printf("DESTINATION  NODE");
     printf("                      NEXT  MODULE");
     printf("\n   #          NODE     MD'L                         ");
     printf("(LOCAL)          ");
     printf("\n\n         #     ST    #    ST    #    ST POS'N  #   ST ");
     printf("POS'N  #   ST  POS'N  #    ST   NODE ST   POS'N\n\n");
     printf(" ( 0");
     for (i = 1; i < MAX_COLUMNS; ++i)
          printf("%5d", i);
     printf(" )");
     printf("\n\n\n");
     for (i = 0; i < nodes; ++i)      {
          printf("\n\n");
          printf("%4d", i);
          for (j = 1; j < MAX_COLUMNS; ++j)
               if (j == 1 || j == 5 || j == 8 || j == 11 || j == 16)
                    printf("%5o", POSITION[i][j]);
               else
                    printf("%5d", POSITION[i][j]);
                                      }

     printf("\n");
}


 save()

/* This function saves current global parameters prior to exit. It is */
/* called by the main network control program.                        */

{
     int i;
     int j;

     outfile = fopen("network.save", "w");
     fprintf(outfile, "%d\n", nodes);
     fprintf(outfile, "%d\n", max_modules);
     fprintf(outfile, "%d\n", max_numbers);
     fprintf(outfile, "%d\n", cur_modules);
     fprintf(outfile, "%d\n", start);
     fprintf(outfile, "\n\n\n\t\t                          ");
     fprintf(outfile, "CONNECTIVITY MATRIX\n\n");
     fprintf(outfile, "   0");
     for (i = 1; i < MAX_COLUMNS; ++i)
          fprintf(outfile, "%5d", i);
     fprintf(outfile, "\n\n");
     for (i = 0; i < nodes; ++i)     {
          fprintf(outfile, "\n\n");
          fprintf(outfile, "%4d", i);
          for (j = 1; j < MAX_COLUMNS; ++j)
               if (j == 1 || j == 5 || j == 8 || j == 11 || j == 16)
                    fprintf(outfile, "%5o", POSITION[i][j]);
               else
                    fprintf(outfile, "%5d", POSITION[i][j]);
                                        }
     for (i = nodes; i < max_numbers; ++i)     {
          fprintf(outfile, "\n\n");
          fprintf(outfile, "%4d %5o", i, POSITION[i][1]);
                                                   }

     fprintf(outfile, "\n\n\n\t\t\t\t");
     fprintf(outfile, "PTR_ARRAY");
     for (i = 0; i < 6; ++i)          {
          fprintf(outfile, "\n\n");
          for (j = 0; j < 2; ++j)
               fprintf(outfile, "%5d", PTR_ARRAY[i][j]);
                                          }
     fclose(outfile);
}


restore()

/* This function restores current parameters at the beginning of      */
/* simulation run.  This way global parameters can be preserved from  */
/* one run until the next. It's called by the main network ctrl prgm. */

{
     int i;
     int j;
     int skip1[20];
     int skip2[20];

     infile = fopen("network.save", "r");
     fscanf(infile, "%d\n", &nodes);
     fscanf(infile, "%d\n", &max_modules);
     fscanf(infile, "%d\n", &max_numbers);
     fscanf(infile, "%d\n", &cur_modules);
     fscanf(infile, "%d\n", &start);
     fscanf(infile, "%*s %*s\n", skip1, skip2);

     for (i = 0; i < MAX_COLUMNS; ++i)
          fscanf(infile, "%*5d", &j);
     for (i = 0; i < nodes; ++i)
          for (j = 0; j < MAX_COLUMNS; ++j)

               if (j == 1 || j == 5 || j == 8 || j == 11 || j == 16)
                    fscanf(infile, "%5o", &POSITION[i][j]);
               else
                    fscanf(infile, "%5d", &POSITION[i][j]);

     for (i = nodes; i < max_numbers; ++i)     {
          fscanf(infile, "%4d", &POSITION[i][0]);
             fscanf(infile, "%5o", &POSITION[i][1]);
                                                 }

     fscanf(infile, "%*s\n", skip1);
     for (i = 0; i < 6; ++i)
          for (j = 0; j < 2; ++j)
               fscanf(infile, "%5d", &PTR_ARRAY[i][j]);

     fclose(infile);

}


clear_failures()

/* This function restores failed nodes and modules to the current      */
/* network topology.  It is called by the main network control prgm.   */

{
     int j;

     for (j = 0; j < start; ++j)     {
          POSITION[j][2] = 0;
          POSITION[j][4] = 0;
          POSITION[j][6] = 0;
```

```
            POSITION[j][9] = 0;
            POSITION[j][12] = 0;
            POSITION[j][15] = 0;
            POSITION[j][17] = 0;
                                          }
    }


    snapshot()

    /* This function saves in a file the current network topology of the  */
    /* simulation module upon request.  Called by main network ctl prgm.  */

    {
         int i;
         int j;
         int c;

    repeat15: printf("\nDo you want a snapshot ");
              printf("of the current topology?    ");

    ignore15: c = getchar();
          if  (c == YES)     {
              snapout = fopen("network.snap", "a");
              fprintf(snapout, "\n\n\n\n\n\n\n");
              fprintf(snapout, "\n\nThe current number of ");
              fprintf(snapout, "modules is %4d.", cur_modules);
              fprintf(snapout, "\n\nThe number of nodes is %4d.", nodes);
              fprintf(snapout, "\n\n\n\n\n\t\t\t\t\t");
              fprintf(snapout, "CONNECTIVITY MATRIX");
              fprintf(snapout, "\n\n");
              fprintf(snapout, "POS'N     HOME     HOME");
              fprintf(snapout, "           DESTINATION  NODE");
              fprintf(snapout, "\t\t\tNEXT MODULE");
              fprintf(snapout, "\n  #       NODE      ");
              fprintf(snapout, "MD'L                     (LOCAL)           ");
              fprintf(snapout, "\n\n     #    ST   #    ST   #    ST");
              fprintf(snapout, " POS'N  #   ST POS'N   #   ST  POS'N");
              fprintf(snapout, "  #   ST  NODE  ST  POS'N\n\n");
              fprintf(snapout, " ( 0");
              for (i = 1; i < MAX_COLUMNS; ++i)
                   fprintf(snapout, "%5d", i);
                   fprintf(snapout, " )");
                   fprintf(snapout, "\n\n\n");
                   for (i = 0; i < nodes; ++i)     {
                       fprintf(snapout, "\n\n");
                       fprintf(snapout, "%4d", i);
                       for (j = 1; j < MAX_COLUMNS; ++j)

                       if (j == 1 || j == 5 || j == 8 || j == 11 || j == 16)
                            fprintf(snapout, "%5o", POSITION[i][j]);
                        else
                            fprintf(snapout, "%5d", POSITION[i][j]);
```

```
                                            }

              fclose(snapout);
                        }
     else if (c == CR)
          goto ignore15;

     else if (c != NO)     {
          printf("\nThat was not a valid response!\n");
          goto repeat15;
                         }
}


simulate()

/* This function is the major control program for traffic simulation  */
/* on the network established in the first part of this program.      */
{
     int temp;

     printf("\nWhat speed would you like to assign to the links?     ");
     scanf("%d", &RATE);
     printf("\nWhat is the mean message size (in characters)?        ");
     scanf("%d", &mean_size);
     printf("\nWhat is the mean message interarrival ");
     printf("time (in msecs)?      ");
     scanf("%d", &IAT);
     printf("\nHow many messages should be delivered ");
     printf("before gathering statistics?     ");
     scanf("%d", &STABILIZE);
repeat6: printf("\nHow many messages should be used ");
     printf("for gathering statistics?     ");
     scanf("%d", &max_msgs);

     if (max_msgs == 0)     {
          printf("\nThat is not a valid request!\n");
          goto repeat6;
                          }

     initialize();
again:  min_FEL();
        time_advance();
     adjust_time(min_time);
     if (imevent == 1)     {
          arrival(min_row);
          if (stop_it == 1)
               goto quit;
                         }
     else     {
          departure(min_row);
          if (stop_it == 1)
               goto quit;
```

```
                }

     more_events();
     temp = no_done - STABILIZE;
     if (temp < max_msgs)
          goto again;
     else      {
          clk_last_msg = CLOCK;
          printf("\n\nTotal messages generated was %d.\n\n", msg_no);
          address_cnt();       /* write to file dest'n address count  */
          graphs();            /* write to file for final graphs      */
quit:     if (stop_it == 0)
               reports();
                }
}


initialize()

/* This function initializes the work areas for the beginning of the  */
/* simulation and schedules the first arrival at each server in the   */
/* network. It is called by the simulate function.                    */

{
     int l;                          /* used to set svc numbers         */
     int i;                          /* used to access rows             */
     int j;                          /* used to access columns          */
     int k;                          /* used to access individual rows */

     CLOCK = 0;
     msg_no = 0;
     min_time = INFINITY;
     location = -1;
     no_done = 0;
     no_killed = 0;
     dirl= 0;                   /* set up matrix needed to find the DF */
     total_hops = 0;
     resp_time = 0;
     max_path = 0;
     stop_it = 0;
     ave_q_length = 0;

     for (i =0; i < LARGEST; ++i)     {
          j = 0;
          while (j < 40)     {
               if (j == 0)
                    MSG_AREA[i][j] = i;
               else if (j == 12)
                    MSG_AREA[i][j] = 12;       /* rt trace is empty   */
               else if (j == 1 || j == 10 || j == 11)
                    MSG_AREA[i][j] = 0;
               else
                    MSG_AREA[i][j] = 999;      /* used to end trace   */
                ++j;
                         }
                                  }
          i = 0;
          while (i < (nodes * 4))     {
               l = 1;
               for (k = i; k < (i + 4); ++k)      {
                    j = 0;
                    while (j < 60)       {
                         if (j == 0)                   /* set row number     */
                              NODE_AREA[k][j] = k;
                         else if (j == 1)                /* set node number  */
                              NODE_AREA[k][j] = (i/4);
                         else if (j == 2)     {          /* set svr#         */
                              NODE_AREA[k][j] = l;
                              ++l;
                                        }
                         else if (j == 5)              /* show server empty */
                              NODE_AREA[k][j] = INFINITY;
                         else if (j == 7)                /* show queue empty */
                              NODE_AREA[k][j] = 8;
                         else
                              NODE_AREA[k][j] = 0;

                         ++j;
                                   }
                                            }
               i = k;
                                   }

          for (i = 1; i <= cur_modules; ++i)    {
               j = 0;
               while (j < 3)     {
                    if (j == 0)
                         GFEL[i][j] = i;
                    else
                         GFEL[i][j] = 0;
                    ++j;
                              }
                                             }

          for (i = 0; i < LARGEST; ++i)          /* set to "0" STATS matrix */
               for (j = 0; j < 5; ++j)
                    if (j == 0)
                         STATS[i][j] = i;       /* save node_area row #    */
                    else
                         STATS[i][j] = 0;

     /* The following code initializes the destination address count.      */

          for (i = 0; i < nodes; ++i)      {
               FREQ[i][0] = i;
               FREQ[i][1] = 0;
```

```
                                    }

/* The next code schedules the first msg for each server of each nd.  */

     i = 0;
     while (i < (nodes * 4))     {
          k = i/4;                   /* used to access rows of POSITION */
          if (POSITION[k][2] == 0)           /* has this node failed? */
               NODE_AREA[i][4] = next_msg();     /* get first arrival */
           else             {
               for (j = i; j < i + 4; ++j)
                    NODE_AREA[j][4] = INFINITY;

               i = j - 1;                     /* skip past failed nodes  */
                              }
          ++i;
                                   }
}


time_advance()

/* This function advances the clock to the beginning of the next      */
/* event and sets imevent for arrival or departure. Sets min_time and */
/* min_row. It is called by the simulate function.                    */

{
     int i;

     min_time = INFINITY;
     for (i = 1; i <= cur_modules; ++i)
          if (GFEL[i][1] <= min_time)     {
               min_time = GFEL[i][1];
               min_row = GFEL[i][2];
                                          }
     if (NODE_AREA[min_row][4] == min_time)
          imevent = 1;
     else
          imevent = 2;
     CLOCK = CLOCK + min_time;

}


adjust_time(time)

/* This function adjusts downward the times of all other activities.  */
/* It is called by the simulate function.                             */

int time;                                   /* contains min_time value   */

{
     int k;
     int j;
     int temp;

     for (k = 0; k < (nodes * 4); ++k)      {
          j = k/4;

/* Adjust times of active nds only, do not bother svrs of failed nds. */

          if (NODE_AREA[k][4] > 0 && POSITION[j][2] == 0) {
               temp = (NODE_AREA[k][4] - time);
               NODE_AREA[k][4] = temp;
                                                            }

          if (NODE_AREA[k][5] > 0 && POSITION[j][2] == 0)     {
               temp = (NODE_AREA[k][5] - time);
               NODE_AREA[k][5] = temp;
                                                                 }
                                            }

     for (k = 0; k < nodes * 4; ++k)
          if ((no_done >= STABILIZE) && (NODE_AREA[k][3] == 1))
               STATS[k][2] = STATS[k][2] + time; /* update svr busy tm */
}


arrival(m)

/* This function handles newly generated messages as they arrive at a */
/* nd.  It calls a function to set initial parameters in the msg_area */
/* matrix.  This function is not called for transient messages since  */
/* they already have key parameters set.  Transient and new arrivals  */
/* are both processed after arrival at a node by the queue function.  */
/* Arrival is called by the simulate function.                        */

int m;                    /* points to min_row in node_area array     */

{
     int place;          /* rename of location for this func          */

     place = set_params(m);
     if (stop_it == 0)      {
          NODE_AREA[m][4] = next_msg();      /* generate next arrival  */
          source_q(m, place);        /* ensure msg starts at right svr */
                                 }
}


 destn_node(x,l)

/* This function terminates a message at its final dest'n node upon   */
/* arrival. It is called by the stage_it function.                    */

int x;                    /*  contains the destination node number    */
int l;                      /*  points to the msg posn in msg_area      */

{

     int i;
     int temp;

     temp = MSG_AREA[l][12];                     /* update route trace  */
     ++temp;
     MSG_AREA[l][temp] = x;
     MSG_AREA[l][12] = temp;

/*  printf("\n\nNode %d has completed delivery", MSG_AREA[l][13]);     */
/*  printf(" to destination node %d.", x);                              */

     MSG_AREA[l][9] = CLOCK;
     ++no_done;
     if ((no_done > STABILIZE) && (no_done <= max_msgs + STABILIZE))
                                        /* has network gone past startup  */
          save_stats(l);

     else                     {
          MSG_AREA[l][1] = 0;            /* show row is now available */
          for (i = 2; i < 40; ++i)
               if (i == 12)
                    MSG_AREA[l][i] = 12;      /* reset trace pointer */
               else if (i == 10 || i == 11)
                    MSG_AREA[l][i] = 0;
               else
                    MSG_AREA[l][i] = 999;      /* clear for reuse  */
                              }
}


 more_events()

/* This function checks the future events list for each nd to see if  */
/* other events are scheduled to occur simultaneously with the         */
/* imminent event. It is called by the simulate function.              */

{
     int l;

     for (l = 0; l < (nodes * 4); ++l)
          if ((NODE_AREA[l][4] == 0) && (l != min_row))
               arrival(l);

     for (l = 0; l < (nodes * 4); ++l)
          if ((NODE_AREA[l][5] == 0) && (l != min_row))
               departure(l);

}
```

```
min_FEL()

/* This function computes the new imminent event at each nd & places  */
/* the min time from each module in the global future events list.    */
/* it is called by the simulate functions.                            */

{
     int i;
     int j;
     int k;
     int temp1;
     int temp2;

     for (i = 0; i < (nodes * 4); ++i)     {
          NODE_AREA[i][6] = NODE_AREA[i][4];
          if (NODE_AREA[i][5] < NODE_AREA[i][6])
               NODE_AREA [i][6] = NODE_AREA[i][5];
                                           }

/* This code updates the global future events lists. It finds the     */
/* minimum FEL value of the 16 servers in each module, &  places that */
/* value in the global FEL.                                           */

     k = 1;
     i = 0;
     while (i < (nodes * 4))     {
          temp1 = INFINITY;
          for (j = i; j < (i + 16); ++j)
               if (NODE_AREA[j][6] < temp1)     {
                    temp1 = NODE_AREA[j][6];
                    temp2 = j;
                                                }
          GFEL[k][1] = temp1;
          GFEL[k][2] = temp2;
          i = j;
          ++k;
                                 }
}


departure(m)

/* This function simulates the departure of a message from the server */
/* (link).  Upon departure, it sets the message up at the receiving   */
/* end for further processing, gives the next waiting message the svr */
/* if one is waiting, or sets the svr to idle if no msgs are queued   */
/* for the server.  It is called by the simulate function.            */

int m;                     /* points to min_row in node_area array    */

{
     int i;
     int j;
     int temp1;
     int temp2;
     int length;

     i = m;
     stage_it(i, NODE_AREA[i][8]);            /* set up msg at recvr     */
     if (NODE_AREA[i][7] == 8)     {          /* is the queue empty      */
          NODE_AREA[i][3] = 0;                /* set svr to idle         */
          NODE_AREA[i][5] = INFINITY;         /* show no departures      */
                                   }

/* This part of the function gives the next msg in the queue the svr, */
/* sets it's departure time on the server's FEL, and updates the q.   */

     else          {
          temp2 = NODE_AREA[i][9];         /* get next msg from queue  */
          for (j = 9; j < 59; ++j)
               NODE_AREA[i][j] = NODE_AREA[i][j + 1];
          temp1 = NODE_AREA[i][7];
          -- temp1;
          NODE_AREA[i][7] = temp1;
          length = MSG_AREA[temp2][3];
          NODE_AREA[i][5] = SERVICE_TIME;
          NODE_AREA[i][8] = temp2;
                    }
}


link_node(i, j)

/* This function finds the next link node in the path of a message as */
/* it is routed to it's destination. This link node may be within the */
/* same module or contained in the "link module" for the node.  It is */
/* called by the stage_it and source_q functions.                     */

int i;                             /* points to node just arrived at   */
int j;                             /* points to msg posn in msg_area   */

{
     int SF;                       /* the spiral flag                  */
     int DF;                       /* the direction flag               */
     int gone;                     /* a flag when a mdl is cut-off     */
     int temp;
     int temp1;
     int temp2;
     int temp3;
     int temp4;                    /* contains module nmbr of cur node */
     int flip;                     /* use to return to again as needed */
     int server1;                  /* points to link node at svr #1    */
     int server2;                  /* points to link node at svr #2    */
     int server3;                  /* points to link node at svr #3    */
     int server4;                  /* points to link node at svr #4    */
     temp = i;                      /* set to node just arrived at       */
     temp4 = POSITION[temp][3];     /* get current module number         */
again: SF = MSG_AREA[j][4];
       SF = SF * 2;                 /* shift one bit left                */
     DF = MSG_AREA[j][5];
     temp2 = SF + DF;               /* logical "or" function             */
     flip = 0;
     gone = 0;

/*  The following test finds and queues at the link node.               */

     temp3 = POSITION[temp][5];
     temp1 = temp3 & 03;
     server1 = temp1;

     temp3 = POSITION[temp][8];
     temp1 = temp3 & 03;
     server2 = temp1;

     temp3 = POSITION[temp][11];
     temp1 = temp3 & 03;
     server3 = temp1;

     if (server1 == temp2)                      /* svr1 link node      */
          if (POSITION[temp][6] == 0)
               queue(temp * 4, j);              /* queue at server #1  */

          else if (MSG_AREA[j][6] == POSITION[temp][7]) /* destn dead */
               kill_it(POSITION[temp][7], temp, j); /* can't make it! */

          else      {
               gone = flip_it(temp, temp4, j);
               if (gone == 0)
                    flip = 1;
                    }

     else if (server2 == temp2)
          if (POSITION[temp][9] == 0)
               queue(temp * 4 + 1, j);         /* queue at server #2   */

          else if (MSG_AREA[j][6] == POSITION[temp][10]) /* dest dead */
               kill_it(POSITION[temp][10], temp, j);

          else      {
               gone = flip_it(temp, temp4, j);
               if (gone == 0)
                    flip = 1;
                    }

     else if (server3 == temp2)
          if (POSITION[temp][12] == 0)
               queue(temp * 4 + 2, j);          /* queue at server #3 */
          else if (MSG_AREA[j][6] == POSITION[temp][13]) /* dest dead */
               kill_it(POSITION[temp][13], temp, j);

          else      {
               gone = flip_it(temp, temp4, j);
               if (gone == 0)
                    flip = 1;
                    }

     else                                   /* the case when svr4 = temp2 */

          if (POSITION[temp][17] == 0)
               queue(temp * 4 + 3, j);           /* queue at server #4 */

          else if (MSG_AREA[j][6] == POSITION[temp][18]) /* dest dead */
               kill_it(POSITION[temp][18], temp, j);

          else      {
               gone = flip_it(temp, temp4, j);
               if (gone == 0)
                    flip = 1;
                    }

     if (flip == 1)
          goto again;          /* return to find alternate link node */

}


flip_it(cur_node, action_mdl, l)

/* This function changes spiral and direction flags to route messages */
/* around failed nodes and modules.  Called by link_node function.    */

int cur_node;                /* contains node just arrived at          */
int action_mdl;              /* contains module number of failed node  */
int l;                       /* points to row in msg_area matrix       */

{

     int gone;            /* used to signal that a mdl is unreachable */
     int temp;
     int source_mdl;

     gone = 0;
     if (MSG_AREA[l][13] == 999)
     if (action_mdl == MSG_AREA[l][10])        /* both spirals cutoff */
          source_mdl = (cur_node / 4) + 1;
     else
          source_mdl = (MSG_AREA[l][13] / 4) + 1; /* get source mdl # */

     if (action_mdl == MSG_AREA[l][10]) /*  changed SF/DF once this   */
                              /* mdl which means both spirals are cut  */
          if (MSG_AREA[1][11] == 0)     {    /* 0 means SF but not DF */
                    /* yet, so chg direct'n retracing previous path   */

               MSG_AREA[1][11] = MSG_AREA[1][11] + 1;
               temp = MSG_AREA[1][5];            /* chg direction flag */
               if (temp == 0)
                    MSG_AREA[1][5] = 1;
               else
                    MSG_AREA[1][5] = 0;
                  temp = MSG_AREA[1][4]; /* reset SF to retrace path  */
               if (temp == 0)
                    MSG_AREA[1][4] = 1;
               else
                    MSG_AREA[1][4] = 0;
                                        }

          else if (MSG_AREA[1][11] == 1)     { /* if 1, already tried */
                 /* direction chg on one spiral, so try same direct'n */
                                             /* on the other spiral */

               MSG_AREA[1][11] = MSG_AREA[1][11] + 1;
               temp = MSG_AREA[1][4];             /* chg spiral flag   */
               if (temp == 0)
                    MSG_AREA[1][4] = 1;
               else
                    MSG_AREA[1][4] = 0;
                                             }

          else
               gone = unreachable(cur_node, source_mdl, 1);

/* In the above case, spiral changes have occured and both directions */
/* have been attempted on both spirals. Can't get to destination.     */

     else          {         /* one spiral is out, chg SF to the other */
          MSG_AREA[1][10] = action_mdl;
          temp = MSG_AREA[1][4];
          if (temp == 0)
               MSG_AREA[1][4] = 1;
          else
               MSG_AREA[1][4] = 0;
                    }

     return(gone);

}


queue(i, j)

/* This function assigns the server if idle, or queues the message if */
/* the server is busy upon arrival of the message.  It's called by    */
/* the stage_it,link_node, and source_q functions.                    */

int i;                        /* points to row of next activity       */
int j;                        /* points to msg place in msg_area array */

{
     int temp;
     int length;

     temp = MSG_AREA[j][12];                     /* update route trace  */
     ++temp;
     MSG_AREA[j][temp] = NODE_AREA[i][1];
     MSG_AREA[j][12] = temp;

     if (NODE_AREA[i][3] == 0)     {           /* Is the server avail  */
          NODE_AREA[i][3] = 1;
          NODE_AREA[i][8] = j;                 /* set link to msg_area */
          length = MSG_AREA[j][3];
          NODE_AREA[i][5] = SERVICE_TIME;      /* set departure time   */
                                    }
     else      {
          temp = NODE_AREA[i][7];
          ++temp;
          if (temp > 59)      {
               printf("\n\nNo more queue slots are available at node ");
               printf("%d, server number", NODE_AREA[i][1]);
               printf(" %d.", NODE_AREA[i][2]);
               stop_it = 1;
                              }

          else                {
               NODE_AREA[i][temp] = j;              /* queue at server  */
               NODE_AREA[i][7] = temp;

               temp = NODE_AREA[i][7] - 8;
               if (no_done >= STABILIZE)
                    if (STATS[i][1] < temp)         /* update max_q len */
                         STATS[i][1] = temp;
                              }
               }
}


save_stats(row)

/* This function saves pertinent information on messages for stats     */
/* analysis.  It also releases the message work area for reuse, and    */
/* updates total_hops thus far for all msgs that have been delivered.  */
/* It is called  by the destination node  function when the final      */
/* destination is reached.                                              */

 int row;                               /* points to row to be saved   */
 {
```

```
          int i;
          int temp;

          savefile = fopen("statistics", "a");

          if ((no_done - STABILIZE) == 1)
     /*                   {                                                   */
                 clk_first_msg = CLOCK;

     /*  The following code is bypassed to reduce disk storage space.         */
     /*  The code is used to get a trace of messages used for statistics.     */
     /*                                                                        */
     /*            fprintf(savefile, "\n\n\n        MSG#  SIZE  SF  DF ");*/
     /*            fprintf(savefile, "TO-ND MDL SEND  REC'V  CS  CD");    */
     /*            fprintf(savefile, " T-PTR\t---- MESSAGE TRACE ----\n");*/
     /*            fprintf(savefile, "\t\t\t\t\t\t (mdl #s) \n");         */
     /*                          }                                        */
     /*    fprintf(savefile, "\n");                                       */
     /*    i = 2;                                                         */
     /*    temp = MSG_AREA[row][i];                                       */
     /*    if (temp != 999)                                               */
     /*    fprintf(savefile, "%5d. ", no_done);                           */
     /*                                                                   */
     /*    while (temp != 999)               {                            */
     /*         if (i == 3 || i == 8 || i == 9)                           */
     /*              fprintf(savefile, "%7d", temp);                      */
     /*         else                                                      */
     /*              fprintf(savefile, "%4d", temp);                      */
     /*         ++i;                                                      */
     /*         temp = MSG_AREA[row][i];                                  */
     /*                                 }                                 */
     /*                                                                   */

          i = MSG_AREA[row][6];          /* prepare to update dest'n address */
          FREQ[i][1] = FREQ[i][1] + 1;

          size_graph(row);               /* update stats to plot size        */

          i = 14;
          temp = MSG_AREA[row][i];
          while (temp != 999)     {
               ++total_hops;             /* update total_hops                */
               ++i;
               temp = MSG_AREA[row][i];
                                  }
          temp = i - 14;                         /* find path length this msg */
          if (temp > max_path)
               max_path = temp;                  /* update max_path length    */

          resp_time = resp_time + (MSG_AREA[row][9] - MSG_AREA[row][8]);

          mean_queue();                          /* total ave_q, all q's      */
          fclose (savefile);

          MSG_AREA[row][1] = 0;                  /* show row is now available */
          for (i = 2; i < 40; ++i)
               if (i == 12)
                    MSG_AREA[row][i] = 12;      /* reset trace pointer       */

               else if (i == 10 || i == 11)
                    MSG_AREA[row][i] = 0;

               else
                    MSG_AREA[row][i] = 999;             /* clear for reuse   */
     }


     set_params(m)

     /* This function assigns the newly arrived message a number, sets the */
     /* destination address, message size, spiral and direction flags, and */
     /* other key parameters.  It is called by the arrival function.       */

     int m;                        /* points to min_row in node_area array */

     {
          int loop;                /* used to find free row in msg_area    */
          int j;

          ++msg_no;
          loop = 0;
     free1:  ++location;
          if (location == LARGEST)     {
               location = 0;
               ++loop;
               if (loop == 2)     {
                    printf("\n\nNo more message spaces available!");
                    stop_it = 1;
                    j = location;
                    goto quit;       /* out of cycle quit, no more spaces */
                              }
                                   }
          j = location;                    /* used to save typing longer var */
          if (MSG_AREA[j][1] != 0)
               goto free1;

          MSG_AREA[j][0] = j;
          MSG_AREA[j][1] = 1;                      /* show row is in use     */
          MSG_AREA[j][2] = msg_no;
          MSG_AREA[j][3] = size();                    /* set new msg size    */
          MSG_AREA[j][6] = address(m);          /* set the  dest'n address */
          MSG_AREA[j][7] = ((MSG_AREA[j][6] / 4) + 1); /* set destn module */
          MSG_AREA[j][8] = CLOCK;
          set_flag(m, j);
```

```
quit:  return(j);

}


stage_it(x,y)

/* This function determines whether or not a message has reached it's */
/* destination module and calls the appropriate functn to process it. */
/* It is called by the departure function.                             */

int x;                          /* points to min row in node_area array  */
int y;                          /* points to msg locat'n msg_area array  */

{
     int temp1;
     int temp2;                 /* points to node just arrived at        */
     int server;                /* contains server no. just vacated      */

     temp1 = NODE_AREA[x][1];   /*  get the minimum node number          */
     server = NODE_AREA[x][2];  /* get svr# of node just departed        */

/* The next if-else stmts find arrived at nd based on svr just left.  */

     if (server == 1)
          temp2 = POSITION[temp1][7];

     else if (server == 2)
          temp2 = POSITION[temp1][10];

     else if (server == 3)
          temp2 = POSITION[temp1][13];

     else
          temp2 = POSITION[temp1][18];

        if (MSG_AREA[y][7] == POSITION[temp2][3])   /* dest'n module  */

          if (MSG_AREA[y][6] == temp2)                  /* dest'n node  */
               destn_node(temp2,y);         /* dest'n node is reached */

          else if (MSG_AREA[y][6] == POSITION[temp2][7])
               if (POSITION[temp2][6] == 0)
                    queue(temp2 * 4, y);      /* take svr #1 to dest'n  */
               else
                    kill_it(POSITION[temp2][7], temp2, y);

          else if (MSG_AREA[y][6] == POSITION[temp2][10])
               if (POSITION[temp2][9] == 0)
                    queue(temp2 * 4 + 1, y);   /* svr #2 to dest'n     */
               else
                    kill_it(POSITION[temp2][10], temp2, y);
          else
               if (POSITION[temp2][12] == 0)
                    queue(temp2 * 4 + 2, y);    /* svr #3 to dest'n    */
               else
                    kill_it(POSITION[temp2][13], temp2, y);

     else
          link_node(temp2,y);                /* queue at link node   */

}


kill_it(k, c, m)

/* This funct'n reports the inability of the network to deliver a msg */
/* to it's final destination due to node failure. It also updates the */
/* counter that keeps track of undelivered messages, and clears the   */
/* message area row for reuse. Called by stage_it, link_node and      */
/* source_q functions.                                                */

int k;                          /* contains number of failed node     */
int m;                          /* contains min row in msg_area matrix */
int c;                          /* contains current node number       */

{

     int i;
     int temp;                     /* contains source node number      */

     if (MSG_AREA[m][13] == 999)
          temp = c;
     else
          temp = MSG_AREA[m][13];

/*   The next few lines were bypassed to reduce program run time.     */

/* printf("\n\nMessage number %d from source node ", MSG_AREA[m][2]); */
/* printf("%d to destination node %d", temp, MSG_AREA[m][6]);         */
/* printf(" \nis killed at node %d because ");                        */
/* printf("node %d has failed!", c, k);                               */

     ++no_killed;
     MSG_AREA[m][1] = 0;                 /* show row is now available */
     for (i = 2; i < 40; ++i)
          if (i == 12)
               MSG_AREA[m][i] = 12;
          else if (i == 10 || i == 11)
               MSG_AREA[m][i] = 0;
          else
               MSG_AREA[m][i] = 999;              /* clear for reuse */
}
```

```
unreachable(cur_node, source_mdl, l)

/* Unreachable reports the inability of the network to deliver a msg  */
/* to it's final destination due to the fact that the netwk is cut in */
/* two. In the case when the network is cut, without this function,   */
/* messages will loop back and forth between the 2 distant-most mdls  */
/* in the contained subloop.  It is called by the flip_it function.   */

int cur_node;                  /* points to the current node of msg   */
int source_mdl;                /* the mdl where looping msg started   */
int l;                         /* msg position in msg_area array      */

{
     int temp;
     int i;
     int gone;                 /* flag used when mdl is unreachable   */

     if (MSG_AREA[l][13] == 999)
          temp = cur_node;
     else
          temp = MSG_AREA[l][13];

/* The next few lines of code was skipped to reduce program run time. */

/* printf("\n\nMessage number %d from source ");                      */
/* printf("module ", MSG_AREA[l][2]);                                 */
/* printf("%d to destination module %d", source_mdl, MSG_AREA[l][7]); */
/* printf(" \nis killed at node %d because module ", temp);           */
/* printf("%d is inaccessible to");                                   */
/* printf(" module %d!", MSG_AREA[l][7], source_mdl);                 */

     ++ no_killed;
     MSG_AREA[l][1] = 0;                 /* show row is now available */
     for (i = 2; i < 40; ++i)
          if (i == 12)
               MSG_AREA[l][i] = 12;
          else if (i == 10 || i == 11)
               MSG_AREA[l][i] = 0;
          else
               MSG_AREA[l][i] = 999;          /* clear for reuse      */
     gone = 1;

     return(gone);
}


set_flag(m, l)

/* Set_flag sets spiral and direction flags for newly arrived msgs.   */
/* It is called by the set_params function.                           */

int m;                        /* points to min_row in node_area array */
int l;                          /*  points to location in msg_area array */

{
     int temp1;
     int temp2;

     temp1 = MSG_AREA[l][6];
     temp2 = POSITION[temp1][1] & 003;        /* pick off lower 3 bits  */

     if ((temp2 == 0) | (temp2 == 1))          /* top spiral             */
          MSG_AREA[l][4] = 0;
     else
          MSG_AREA[l][4] = 1;                  /* bottom spiral          */

     if (dir1 == 0)                      /* set up matrix to find DF    */
          dir();
     DF_set(m, l);

}


double rndm()

/* This function returns a random number between 0 and 1. Called by   */
/* the size and next_msg function.                                     */

{
     float normalize;
     unsigned int y;
     double temp;

     y = rand();
     normalize = pow(2.0, 31.0) - 1;
     temp = y / normalize;
     return((double)temp);
}


size()

/* This function calculates a random size for the next message, based */
/* on an exponential distribution.  Called by the set_params funct'n. */

{
     float length;
     double rndm();

again:  length = (-mean_size * log(rndm())) * 8.0;
     if (SERVICE_TIME < 1)
          goto again;

     return(length);
}
```

```
address(m)

/* This function generates the address for newly arrived messages. It */
/* is called by the set_params function.                              */

int m;                        /* points to min_row in node_area matrix */
{

     int i;
     unsigned int temp;

next1: temp = rand();
     i = temp % nodes;

     if ((i >= nodes) !! (i == NODE_AREA[m][1]))
          goto next1;
/*        else     {                                                   */
/*  The next three lines were bypassed to reduce program run time.     */

/*        printf("\n\nNode %d is sending to dest'n ", NODE_AREA[m][1]); */
/*        printf("node %d.", i);                                        */
/*                   }                                                  */

        return(i);
}


next_msg()

/* This function calculates and returns the arrival time (in msecs)   */
/* for the next message.  The value returned is selected from an      */
/* exponential probability distribution, which means the message      */
/* arrival pattern follows a poisson arrival process.  Next_msg is    */
/* called by the initialize and arrival functions.                    */

{
     double rndm();
     float time;

next1:  time = -IAT * log(rndm());
     if (time < 1)
          goto next1;

     return (time);
}


write_matrix()

/*  This function prints the network connectivity matrix.         */
```

```
{
     int i;
     int j;

     printf(" ( 0");
     for (i = 1; i < 19; ++i)
          printf("%5d", i);
     printf(" )");
     printf("\n\n\n");
     for (i = 0; i < nodes; ++i)     {
          printf("\n\n");
          printf("%4d", i);
          for (j = 1; j < 19; ++j)
               if (j == 1 !! j == 5 !! j == 8 !! j == 11 !! j == 16)
                    printf("%5o", POSITION[i][j]);
               else
                    printf("%5d", POSITION[i][j]);
                                   }
}


pGFEL()

/*  This function prints the global future events list.                */

{
     int i;
     int j;

     for (i = 1; i <= cur_modules; ++i)     {
          printf("\n\n");
          for (j = 0; j < 3; ++j)
               printf("\t%5d", GFEL[i][j]);
                                            }
}


pNODE_AREA()

/*  This function prints the node_area matrix.                         */

{
     int i;
     int j;

     printf("\n\n\n");
     for (i = 0; i < 10; ++i)     {
          printf("\n\n");
          for (j = 0; j < 28; ++j)
               printf("%5d", NODE_AREA[i][j]);
                                  }
}
```

```
pSTATS()

/*  This function prints the STATS matrix.                              */

{
     int i;
     int j;

     printf("\n\n");
     for (i = 0; i < 64; ++i)     {
          printf("\n");
          for (j = 0; j < 5; ++j)
               printf("%5d", STATS[i][j]);
                                   }
}


dir()

/* This function sets up the work matrix used to find the distance    */
/* from source module to destination mdl. Sets END to dest'n, BEGIN   */
/* to source. It does so by establishing an array of module numbers   */
/* in the order the modules are accessed via the network threading    */
/* pattern. The access pattern is not sequential!  dir() is called by */
/* the set_flag function.                                             */

{
     int temp;
     int k;
     int next;
     int module;

     dir1 = 1;
     temp = 0;
     for (k = 0; k < cur_modules; ++k)     {
          next = POSITION[temp][7];
          temp = POSITION[next][18];
          module = POSITION[next][14];
          DIR[k][0] = k;
          DIR[k][1] = module;
                                          }

}


DF_set(m, l)

/* This function uses the DIR matrix to set the direction flag. It    */
/* insures that the path taken in the shortest in every case.  It is  */
/* called by the set_flag function.                                   */

int m;                        /* points to min_row in node_area array  */
int l;                       /* points to location in msg_area array  */

{
     int k;
     int temp3;
     int temp;
     int temp1;

     temp = NODE_AREA[m][1];            /* get node # of min activity */
     for (k = 0; k < cur_modules; ++k)      {
          if (DIR[k][1] == MSG_AREA[l][7])     /* destination module   */
               END = DIR[k][0];
          if (DIR[k][1] == POSITION[temp][3])  /* source module        */
               BEGIN = DIR[k][0];
                                            }

     temp3 = END - BEGIN;    /* find the distance fm source to dest'n */

/* If the number of modules is odd, then the maximum distance between */
/* any two is an even integer, thus the maximum path length is also   */
/* even (even #/2). So the paths split in exactly half.  When the     */
/* distance is the same in both directions, the algorithm selects     */
/* with probability .5 either of the directions.  This selection is   */
/* made based on whether the destination is to the left or right of   */
/* the source.                                                        */

if (cur_modules % 2 == 1)      {           /* Odd number of modules    */
     temp1 = cur_modules/2;
     if (temp3 <= temp1 && temp3 > 0)
          MSG_AREA[l][5] = 1;              /* go in the left direction */

     else if (temp3 < -temp1)
          MSG_AREA[l][5] = 1;              /* go in the left direction */

     else
          MSG_AREA[l][5] = 0;             /* go in the right direction */
                               }

/* If the number of modules is even, the maximum distance between'em  */
/* is an odd integer, which divided in half yields a remainder term.  */
/* If the mdle distance is the same in both directions, a subroutine  */
/* called "equal" is called to set DF based on the location of the    */
/* S/D nodes on the Source/Destination module.                        */

else            {                         /* Even number of modules!   */
     temp1 = cur_modules/2;

     if (temp3 < temp1 && temp3 > 0)      /* strictly less than case   */
          MSG_AREA[l][5] = 1;             /* go in the left direction  */

     else if (temp3 < -temp1)             /* another < case, still     */
          MSG_AREA[l][5] = 1;             /* go in the left  direction */
     else if ((temp3 > -temp1 && temp3 < 0) !! (temp3 > temp1))
          MSG_AREA[l][5] = 0;             /* go in the right direction */

     else                            /* the case when temp1 equals temp3  */
          equal(m, l);        /* so call "equal" to examine S/D nodes  */
              }
}


equal(m, l)

/* This short function is the last option to setting the DF when the  */
/* distance from source module to destination module is exactly      */
/* equal in both directions.  It is called by the DF_set function     */

int m;                       /* points to min_row in node_area matrix     */
int l;                      /* points to msg location in msg_area matrix  */

{
     if ((NODE_AREA[m][1] % 2 == 0) && (MSG_AREA[l][6] % 2 == 1))
                                                  /* s-even, d-odd   */
               MSG_AREA[l][5] = 0;                /* go to the right */
     else if ((NODE_AREA[m][1] % 2 == 1) && (MSG_AREA[l][6] % 2 == 0))
                                                  /* s-odd, d-even   */
          MSG_AREA[l][5] = 1;                     /* go to the left  */
     else if ((NODE_AREA[m][1] % 2 == 0) && (MSG_AREA[l][6] % 2 == 0))
                                                  /* s & d even      */
          MSG_AREA[l][5] = 1;                     /* go to the left  */
     else
          MSG_AREA[l][5] = 0;                     /* go to the right */
}


pMSG_AREA()

/*  This function prints the message_area matrix.                        */

{
     int m;
     int n;

     printf("\n\n\n");
     for (m = 0; m < 20; ++m)     {
          printf("\n\n");
          for (n = 0; n < 30; ++n)
               printf("%5d", MSG_AREA[m][n]);
                                        }
}


source_q(m, l)

/* This function ensures that newly arrived messages at each node are */
/* placed on the correct output queue from that nd.  Since "new' msgs */
/* arrive at the links of the node, failure to transfer them to the   */
/* proper output queue from that node lengthens the route towards the */
/* destination. This function is called by the arrival function.      */

int m;                      /* points to min_row in node_area array  */
int l;                      /* points to msg_posn in msg_area array  */

{

     int temp1;             /* points to node msg just arrived at    */
     temp1 = NODE_AREA[m][1];   /* get node msg just arrived at      */

     if (MSG_AREA[1][7] != POSITION[temp1][3])      /* dest'n module  */
          link_node(temp1, l);

     else
          if (MSG_AREA[1][6] == POSITION[temp1][7]) /* find dest'n nd */
               if (POSITION[temp1][6] == 0)
                    queue(temp1 * 4, l);      /* svr #1 to destination */
               else
                    kill_it(POSITION[temp1][7], temp1, l);

          else if (MSG_AREA[1][6] == POSITION[temp1][10])
               if (POSITION[temp1][9] == 0)
                    queue(temp1 * 4 + 1, l);      /* svr #2 to dest'n  */
               else
                    kill_it(POSITION[temp1][10], temp1, l);

          else if (MSG_AREA[1][6] == POSITION[temp1][13])
               if (POSITION[temp1][12] == 0)
                    queue(temp1 * 4 + 2, l);      /* svr #3 to dest'n  */
               else
                    kill_it(POSITION[temp1][13], temp1, l);

          else
               printf("\n\nLogic error in source_q function!");
}


address_cnt()

/* This function saves the count of the number of msgs sent to each   */
/* node in the network.  This information is stored in a file called  */
/* destination.  This function is called by the simulate function.    */

{
     int i;

     frequency = fopen("destination", "a");
     fprintf(frequency, "\n\n\n        NODE   NO_MSGS\n\n");

     for (i = 0; i < nodes; ++i)      {
          fprintf(frequency, "\t%d \t%d", FREQ[i][0], FREQ[i][1]);
          fprintf(frequency, "\n");
                                        }

     fclose(frequency);
}


size_graph(row)

/* Size_graphs generates stats used to plot the message size graphs.  */
/* It is called by the save_stats function.                            */

int row;
{
     int step1;
     int step2;
     int temp;

     if (MSG_AREA[row][3] < 2000)
          ++STATS[0][3];

     else if (MSG_AREA[row][3] >= 38000)
          ++STATS[19][3];

     else      {
          step2 = 2;
          for (step1 = 2; step1 < 20; ++step1)
               if((MSG_AREA[row][3] >= 1000 * step2) &&
                    (MSG_AREA[row][3] < 2000 * step1))     {
                         temp = step1 - 1;
                         ++STATS[temp][3];
                         step2 = 2 * step1;
                                                       }
               }
}


graphs()

/* This function saves the numbers used to sketch the graphs of msg    */
/* size and interarrival time distributions.  It moves these values    */
/* from a matrix called STATS into a file called graphs. This funct'n */
/* is called by the simulate function.                                 */

{
     int step1;
     int step2;
     int temp;
     int temp2;

     graphit = fopen("graphs", "a");
     fprintf(graphit, "\n\n\n\t\t MSG SIZE STATS");
```

```
          fprintf(graphit, "\n\n      class#   class range  frequency");
          step2 = 3;
          fprintf(graphit, "\n\n\t1.   ");
          temp2 = STATS[0][3];
          fprintf(graphit, "   0  <  3000\t%d", temp2);

          for (step1 = 2; step1 < 20; ++step1)      {
               temp = step1 - 1;
               fprintf(graphit, "\n\n\t%d.", step1);
               if (step1 < 10)
                    fprintf(graphit, " ");
               temp2 = STATS[temp][3];
               fprintf(graphit, "  %5d < %5d", 1000 * step2, 3000 * step1);
               fprintf(graphit, " \t%d", temp2);
               step2 = 3 * step1;
                                                   }

          fprintf(graphit, "\n\n\t20.  ");
          temp2 = STATS[19][3];
          fprintf(graphit, "    > = 57000\t%d", temp2);
          fprintf(graphit, "\n\n\n\n\n\n\n\n\n");
     }


     mean_queue()

     /* Mean_queue updates the running total used to find the mean queue   */
     /* length at the end of the simulation run. It is called by the save_ */
     /* stats function.                                                     */

     {
          int i;
          int j;
          int k;
          int failed_nodes;
          int temp1;
          int zeroes;
          float temp;
          float temp2;

          k = 0;
          temp = 0;
          temp1 = 0;
          temp2 = 0;
          failed_nodes = 0;
          zeroes = 0;
     /*   zeroes = 0 - 1;                                                   */

          for (i = 0; i < nodes; ++i)
               if (POSITION[i][2] == 1)
                    ++failed_nodes;

          for (i = 0; i < nodes * 4; ++i)     { /* get total q length sum   */
```

```
            j = NODE_AREA[i][7];
            temp1 = j - 8;
/*          k = i/4;                                                    */

/*          if (POSITION[k][2] == 0 && temp1 == 0)                      */
/*               ++zeroes;                   /* count #  of empty queues */
/*         else if (POSITION[k][2] == 0)                                */
                 temp = temp + temp1;
                                       }

      temp1 = (nodes - failed_nodes) * 4;
      temp2 = temp / (temp1 - zeroes);
      ave_q_length = ave_q_length + temp2; /* sum of ave_q length when */
                                           /* each message is delivered */
}


reports()

/* This function calculates and reports for study, results of the     */
/* simulation run.  It is called by the simulate funcion.             */

{
      int i;
      int n;
      int left_msgs;
      int failed_nodes;
      int temp1;
      int j;
      int mean_time;                    /* mean msg tx time per hop    */
      int network_time;                 /* for time til next msg del'd */
      float temp;
      float temp2;
      float m;
      float p;
      float l;
      float k;
      float d;
      float ave_path;
      float ave_delay;                  /* to find ave n/wk resp time  */
      float delay_hop;                  /* average message delay/hop   */

      failed_nodes = 0;
      statsfile = fopen("summary", "a");
      fprintf(statsfile, "\n\n\n\n");
      fprintf(statsfile, "\n\n   \t\t\tSUMMARY OF SIMULATION RESULTS");
      fprintf(statsfile, "\n\n\n");
      fprintf(statsfile, "   \t\tNumber of modules (nodes):\t\t");
      fprintf(statsfile, "%d (%d)", max_modules, nodes);

      j = 0;
      i = 0;
      fprintf(statsfile, "\n\n   \t\tFailed module(s):\n\t\t  ");
```

```
while (i < nodes)     {
     if (POSITION[i][4] == 1)     {
          fprintf(statsfile, "%4d", POSITION[i][3]);
          ++j;
                                        }
     i = i + 4;
                         }
if (j == 0)
     fprintf(statsfile, "    NONE!");

j = 0;
n = 0;
fprintf(statsfile, "\n\n");
fprintf(statsfile, "   \t\tFailed node(s) (including those ");
fprintf(statsfile, "in failed modules):\n\t\t  ");
for (i = 0; i < nodes; ++i)
     if (POSITION[i][2] == 1) {
          fprintf(statsfile, "%4d", i);
          ++j;
          ++failed_nodes;
          ++n;
          if (n > 10)            {
               n = 0;
               fprintf(statsfile, "\n  \t\t  ");
                                   }
                              }
if (j == 0)
     fprintf(statsfile, "    NONE!");

mean_time = (1000 * mean_size * 8) / RATE; /* trns time in msecs */
if (mean_time > (resp_time / max_msgs))
     mean_time = resp_time / max_msgs;

if ((clk_last_msg - clk_first_msg) < mean_time)
     network_time = mean_time;         /* total time it takes to  */
                                        /* deliver max_msgs        */
else
     network_time = clk_last_msg - clk_first_msg;

temp2 = max_msgs;
temp = network_time;
l = temp / temp2;                    /* time 'til nxt msg delivered */
left_msgs = msg_no - (STABILIZE + no_killed + max_msgs);

temp = 0;
for (i = 0; i < nodes * 4; ++i)
     temp = temp + STATS[i][2];    /* total time all svrs busy    */
k = temp / ((nodes - failed_nodes) * 4); /* ave link busy time   */

m = k / network_time;                 /* prob that a link is busy   */
if (m > 1)                            /* this text is needed since  */
     m = 1;                           /* several msgs can arrive in */
                                      /* short period of time       */
```

```
temp1 = 0;
temp = STATS[temp1][1];
for (i = 1; i < nodes * 4; ++i)              {
     if (STATS[i][1] > temp)     {
          temp = STATS[i][1];       /* find the maximum q length  */
          temp1 = i;                /* corresp node/svr it's at   */
                                 }
                                              }
n = temp;                           /* leave it in variable n     */

p = ave_q_length / max_msgs;        /* find average queue length  */

ave_path = total_hops / max_msgs;   /* find average path length   */
temp = resp_time;
temp2 = max_msgs;
ave_delay = temp / temp2;           /* average delay per msg/hop  */

delay_hop = ave_delay / ave_path;       /* find ave delay/msg/hop */

if (delay_hop <= mean_time)     {
     delay_hop = mean_time;
     d = 0;
                                }
else
     d = delay_hop - mean_time;   /* average queueing time/msg    */


fprintf(statsfile, "\n\n   \t\tMean message size:");
fprintf(statsfile, "\t\t\t%d bits", mean_size * 8);
fprintf(statsfile, "\n\n   \t\tLine speed all links:\t\t\t");
fprintf(statsfile, "%d bits/sec", RATE);
fprintf(statsfile, "\n\n   \t\tMean message interarrival ");
fprintf(statsfile, "time:\t\t%d msecs", IAT);
fprintf(statsfile, "\n\n   \t\tTotal messages ");
fprintf(statsfile, "generated:\t\t%d", msg_no);
fprintf(statsfile, "\n\n   \t\tMessages delivered before");
fprintf(statsfile, " stats:\t%d", STABILIZE);
fprintf(statsfile, "\n\n   \t\tMessages used for");
fprintf(statsfile, " statistics:\t\t%d", max_msgs);
fprintf(statsfile, "\n\n  \t\tMessages undelivered due to ");
fprintf(statsfile, "failure(s):\t%d", no_killed);
fprintf(statsfile, "\n\n   \t\tMessages left in network");
fprintf(statsfile, ":\t\t%d", left_msgs);
fprintf(statsfile, "\n\n   \t\tMaximum queue ");
fprintf(statsfile, "length:\t\t\t%d msgs", n);

if (n > 0)     {
     fprintf(statsfile, "\n\n   \t\tNode with max queue:\t\t\t");
     fprintf(statsfile, "%d", NODE_AREA[temp1][1]);
     fprintf(statsfile, "\n\n   \t\tLink with max queue:\t\t\t");
     fprintf(statsfile, "%d", NODE_AREA[temp1][2]);
               }
```

```
    else            {
         fprintf(statsfile, "\n\n   \t\tNode with ");
         fprintf(statsfile, "max queue:\t\t\tNONE");
         fprintf(statsfile, "\n\n   \t\tLink with ");
         fprintf(statsfile, "max queue:\t\t\tNONE");
                    }
    fprintf(statsfile, "\n\n   \t\tMean queue ");
    fprintf(statsfile, "length:\t\t\t%.6f msgs", p);
    fprintf(statsfile, "\n\n   \t\tMaximum path length:\t\t\t");
    fprintf(statsfile, "%d hops", max_path);
    fprintf(statsfile, "\n\n   \t\tMean path length:\t\t\t");
    fprintf(statsfile, "%.4f hops", ave_path);
    fprintf(statsfile, "\n\n   \t\tMean response time per");
    fprintf(statsfile, " message:\t\t%.4f msecs", ave_delay);
    fprintf(statsfile, "\n\n   \t\tMean delay/hop:\t\t\t\t");
    fprintf(statsfile, "%.4f msecs",  delay_hop);
    fprintf(statsfile, "\n\n   \t\tMean transmission time/hop:");
    fprintf(statsfile, "\t\t%d msecs", mean_time);
    fprintf(statsfile, "\n\n   \t\tMean queueing time/hop:");
    fprintf(statsfile, "\t\t\t%.4f msecs", d);
    fprintf(statsfile, "\n\n   \t\tMean link busy time:");
    fprintf(statsfile, "\t\t\t%.4f msecs", k);
    fprintf(statsfile, "\n\n   \t\tProbability of link busy");
    fprintf(statsfile, " (rho):\t\t%.6f", m);
    fprintf(statsfile, "\n\n   \t\tProbability msg does not queue");
    fprintf(statsfile, ":\t\t%.6f", 1 - m);
    fprintf(statsfile, "\n\n\n\n\n\n\n\n\n");

    fclose(statsfile);

}
```